

*A PRACTICAL GUIDE TO DATA
STRUCTURES AND ALGORITHMS
USING JAVA*

Chapman & Hall/CRC
Applied Algorithms and Data Structures Series

Series Editor

Samir Khuller
University of Maryland

Aims and Scopes

The design and analysis of algorithms and data structures form the foundation of computer science. As current algorithms and data structures are improved and new methods are introduced, it becomes increasingly important to present the latest research and applications to professionals in the field.

This series aims to capture new developments and applications in the design and analysis of algorithms and data structures through the publication of a broad range of textbooks, reference works, and handbooks. We are looking for single authored works and edited compilations that will:

- Appeal to students and professionals by providing introductory as well as advanced material on mathematical, statistical, and computational methods and techniques
- Present researchers with the latest theories and experimentation
- Supply information to interdisciplinary researchers and practitioners who use algorithms and data structures but may not have advanced computer science backgrounds

The inclusion of concrete examples and applications is highly encouraged. The scope of the series includes, but is not limited to, titles in the areas of parallel algorithms, approximation algorithms, randomized algorithms, graph algorithms, search algorithms, machine learning algorithms, medical algorithms, data structures, graph structures, tree data structures, and more. We are willing to consider other relevant topics that might be proposed by potential contributors.

Proposals for the series may be submitted to the series editor or directly to:

Randi Cohen

Acquisitions Editor

Chapman & Hall/CRC Press

6000 Broken Sound Parkway NW, Suite 300

Boca Raton, FL 33487

A PRACTICAL GUIDE TO DATA STRUCTURES AND ALGORITHMS USING JAVA

Sally Goldman
Kenneth Goldman

Washington University
Saint Louis, Missouri, U.S.A.



Chapman & Hall/CRC

Taylor & Francis Group

Boca Raton London New York

Chapman & Hall/CRC is an imprint of the
Taylor & Francis Group, an **informa** business

Cover Design by Ben Goldman

Chapman & Hall/CRC
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 2008 by Taylor & Francis Group, LLC
Chapman & Hall/CRC is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works
Printed in the United States of America on acid-free paper
10 9 8 7 6 5 4 3 2 1

International Standard Book Number-13: 978-1-58488-455-2 (Hardcover)

This book contains information obtained from authentic and highly regarded sources. Reprinted material is quoted with permission, and sources are indicated. A wide variety of references are listed. Reasonable efforts have been made to publish reliable data and information, but the author and the publisher cannot assume responsibility for the validity of all materials or for the consequences of their use.

No part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com (<http://www.copyright.com/>) or contact the Copyright Clearance Center, Inc. (CCC) 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Library of Congress Cataloging-in-Publication Data

Goldman, Sally A.

A practical guide to data structures and algorithms using java / Sally A. Goldman and Kenneth J. Goldman.

p. cm.

Includes bibliographical references and index.

ISBN-13: 978-1-58488-455-2 (alk. paper)

1. Java (Computer program language) 2. Data structures (Computer science) I. Goldman, Kenneth J. II. Title.

QA76.73.J38G589 2007

005.13'3--dc22

2007016305

Visit the Taylor & Francis Web site at
<http://www.taylorandfrancis.com>

and the CRC Press Web site at
<http://www.crcpress.com>

to Julie, Ben, and Mark

Contents

Preface	xxiii
Acknowledgments	xxv
Authors	xxvii

I INTRODUCTION 1

1 Design Principles 3

1.1	Object-Oriented Design and This Book	4
1.2	Encapsulation	5
1.3	Invariants and Representation Properties	6
1.4	Interfaces and Data Abstraction	7
1.5	Case Study on Conceptual Design: Historical Event Collection	8
1.6	Case Study on Structural Design: Trees	10
1.7	Further Reading	13

2 Selecting an Abstract Data Type 15

2.1	An Illustrative Example	16
2.2	Broad ADT groups	19
2.3	Partition of a Set	21
2.4	A Collection of Elements	21
2.5	Markers and Trackers	23
2.6	Positioning and Finding Elements	24
2.6.1	Manually Positioned Collections	26
2.6.2	Algorithmically Positioned Collections	26
2.7	Graphs	28

3 How to Use This Book 35

3.1	Conventions	35
3.2	Parts II and III Presentation Structure	36
3.2.1	ADT Chapters	36
3.2.2	Data Structures	38
3.2.3	Algorithms	40
3.3	Appendices and CD	41

II COLLECTION DATA STRUCTURES AND ALGORITHMS 43

4 Part II Organization 45

5 Foundations 49

5.1	Wrappers for Delegation	50
5.2	Objects Abstract Class	51
5.2.1	Singleton Classes: Empty and Deleted	51
5.2.2	Object Equivalence	51
5.2.3	Object Comparison	53

5.3	Digitizer Interface	54
5.4	Bucketizer Interface	56
5.5	Object Pool to Reduce Garbage Collection Overhead	59
5.6	Concurrency	60
5.7	Iterators for Traversing Data Structures	62
5.8	Locator Interface	63
5.8.1	Case Study: Maintaining Request Quorums for Byzantine Agreement . .	63
5.8.2	Interface	64
5.8.3	Markers and Trackers	65
5.8.4	Iteration Using Locators	67
5.8.5	Iteration Order and Concurrent Modifications	68
5.9	Version Class	70
5.10	Visitors for Traversing Data Structures	71
6	Partition ADT and the Union-Find Data Structure	73
6.1	Partition Element Interface	73
6.2	Selecting a Data Structure	74
6.3	Union-Find Data Structure	74
6.4	Internal Representation	75
6.5	Representation Properties	76
6.6	Methods	77
6.7	Performance Analysis	80
6.8	Case Study: Preserving Locators When Merging Data Structures	81
6.9	Further Reading	83
6.10	Quick Method Reference	87
7	Collection of Elements	89
7.1	Collection Interface	90
7.2	Tracked Collection Interface	92
7.3	ADTs Implementing the Collection Interface	92
7.3.1	Manually Positioned Collections	92
7.3.2	Algorithmically Positioned Untagged Collections	92
7.3.3	Algorithmically Positioned Tagged Ungrouped Collections	93
7.3.4	Algorithmically Positioned Tagged Grouped Collections	94
8	Abstract Collection	95
8.1	Internal Representation	95
8.2	Representation Properties	96
8.3	Methods	96
8.3.1	Trivial Accessors	96
8.3.2	Algorithmic Accessors	97
8.3.3	Representation Mutators	99
8.3.4	Content Mutators	100
8.4	Abstract Locator Inner Class	100
8.5	Visiting Iterator	102
8.6	Performance Analysis	105
8.7	Quick Method Reference	105

9	Positional Collection ADT	107
9.1	Interface	107
9.2	Positional Collection Locator Interface	109
9.3	Terminology	110
9.4	Competing ADTs	110
9.5	Selecting a Data Structure	111
9.5.1	Tradeoffs among Array-Based Data Structures	114
9.5.2	Tradeoffs among List-Based Data Structures	115
9.6	Summary of Positional Collection Data Structures	116
9.7	Further Reading	119
10	Abstract Positional Collection	121
10.1	Abstract Positional Collection	121
10.2	Internal Representation	121
10.3	Quick Method Reference	122
11	Array Data Structure	125
11.1	Internal Representation	126
11.2	Representation Properties	128
11.3	Methods	128
11.3.1	Constructors	128
11.3.2	Trivial Accessors	129
11.3.3	Representation Accessors	130
11.3.4	Algorithmic Accessors	131
11.3.5	Representation Mutators	132
11.3.6	Content Mutators	133
11.3.7	Locator Initializers	139
11.4	Sorting Algorithms	140
11.4.1	Insertion Sort	143
11.4.2	Mergesort	145
11.4.3	Heap Sort	147
11.4.4	Tree Sort	149
11.4.5	Quicksort	149
11.4.6	Radix Sort	155
11.4.7	Bucket Sort	157
11.5	Selection and Median Finding	158
11.6	Basic Marker Inner Class	160
11.7	Marker Inner Class	162
11.8	Performance Analysis	163
11.9	Further Reading	166
11.10	Quick Method Reference	167
12	Circular Array Data Structure	171
12.1	Internal Representation	171
12.2	Representation Properties	173
12.3	Methods	173
12.3.1	Constructors	173
12.3.2	Representation Accessors	174
12.3.3	Representation Mutators	175
12.3.4	Content Mutators	176
12.4	Performance Analysis	181
12.5	Quick Method Reference	183

13	Dynamic Array and Dynamic Circular Array Data Structures	185
13.1	Dynamic Array	185
13.2	Internal Representation	186
13.3	Representation Properties	187
13.4	Methods	187
13.4.1	Constructors	187
13.4.2	Representation Mutators	188
13.4.3	Content Mutators	189
13.5	Performance Analysis	190
13.6	Dynamic Circular Array	191
14	Tracked Array Data Structure	193
14.1	Internal Representation	194
14.2	Representation Properties	197
14.3	Node Inner Class	198
14.4	Tracked Array Methods	198
14.4.1	Constructors	198
14.4.2	Representation Accessors	199
14.4.3	Algorithmic Accessors	199
14.4.4	Representation Mutators	200
14.4.5	Content Mutators	201
14.4.6	Locator Initializers	204
14.5	Wrappers for Sorting Tracked Arrays	205
14.6	Tracker Inner Class	207
14.7	Performance Analysis	213
14.8	Quick Method Reference	214
15	Singly Linked List Data Structure	217
15.1	Internal Representation	217
15.2	Representation Properties	220
15.3	List Item Inner Class	220
15.4	Singly Linked List Methods	221
15.4.1	Constructors and Factory Methods	221
15.4.2	Representation Accessors	222
15.4.3	Algorithmic Accessors	222
15.4.4	Representation Mutators	224
15.4.5	Content Mutators	224
15.4.6	Locator Initializers	231
15.5	Sorting Algorithms Revisited	232
15.5.1	Insertion Sort	232
15.5.2	Mergesort	233
15.5.3	Heap Sort	235
15.5.4	Tree Sort	236
15.5.5	Quicksort	237
15.5.6	Radix Sort	242
15.5.7	Bucket Sort	244
15.6	Selection and Median Finding	244
15.7	Tracker Inner Class	246
15.8	Performance Analysis	251
15.9	Quick Method Reference	254

16 Doubly Linked List Data Structure	257
16.1 Internal Representation	257
16.2 Representation Properties	258
16.3 Doubly Linked List Item Inner Class	259
16.4 Doubly Linked List Methods	259
16.4.1 Constructors and Factory Methods	259
16.4.2 Representation Accessors	260
16.4.3 Algorithmic Accessors	260
16.4.4 Representation Mutators	261
16.5 Performance Analysis	261
16.6 Quick Method Reference	263
17 Buffer ADT and Its Implementation	265
17.1 Internal Representation	265
17.2 Representation Properties	266
17.3 Methods	266
17.3.1 Constructors	266
17.3.2 Trivial Accessors	267
17.3.3 Algorithmic Accessors	268
17.3.4 Content Mutators	268
17.3.5 Locator Initializers	269
17.4 Performance Analysis	270
17.5 Quick Method Reference	270
18 Queue ADT and Implementation	271
18.1 Internal Representation	271
18.2 Methods	271
18.3 Performance Analysis	272
18.4 Quick Method Reference	273
19 Stack ADT and Implementation	275
19.1 Internal Representation	275
19.2 Methods	276
19.3 Performance Analysis	278
19.4 Quick Method Reference	278
20 Set ADT	279
20.1 Case Study: Airline Travel Agent	279
20.2 Interface	280
20.3 Terminology	280
20.4 Hasher Interface	281
20.5 Competing ADTs	282
20.6 Selecting a Data Structure	283
20.7 Summary of Set Data Structures	286
20.8 Further Reading	289
21 Direct Addressing Data Structure	291
21.1 Internal Representation	291
21.2 Representation Properties	293
21.3 Default Direct Addressing Hasher Class	293
21.4 Methods	293

21.4.1	Constructors	293
21.4.2	Trivial Accessors	294
21.4.3	Representation Accessors	294
21.4.4	Algorithmic Accessors	295
21.4.5	Representation Mutators	296
21.4.6	Content Mutators	296
21.4.7	Locator Initializers	297
21.5	Marker Inner Class	298
21.6	Performance Analysis	301
21.7	Quick Method Reference	303
22	Open Addressing Data Structure	305
22.1	Internal Representation	305
22.2	Representation Properties	308
22.3	Default Open Addressing Hasher Class	309
22.4	Open Addressing Methods	310
22.4.1	Constructors	310
22.4.2	Trivial Accessors	311
22.4.3	Representation Accessors	311
22.4.4	Selecting a Hash Function	312
22.4.5	Algorithmic Accessors	313
22.4.6	Representation Mutators	314
22.4.7	Content Mutators	316
22.5	Performance Analysis	317
22.6	Quick Method Reference	318
23	Separate Chaining Data Structure	321
23.1	Internal Representation	322
23.2	Representation Properties	324
23.3	Chain Item Inner Class	325
23.4	Default Separate Chaining Hasher Class	326
23.5	Separate Chaining Methods	326
23.5.1	Constructors	326
23.5.2	Trivial Accessors	328
23.5.3	Algorithmic Accessors	328
23.5.4	Representation Mutators	329
23.5.5	Content Mutators	331
23.5.6	Locator Initializers	333
23.6	Marker Inner Class	334
23.7	Performance Analysis	339
23.8	Quick Method Reference	340
24	Priority Queue ADT	343
24.1	Case Study: Huffman Compression	343
24.2	Interface	345
24.3	Priority Queue Locator Interface	345
24.4	Selecting a Data Structure	346
24.5	Terminology	347
24.6	Competing ADTs	347
24.7	Summary of Priority Queue Data Structures	348
24.8	Further Reading	351

25 Binary Heap Data Structure	353
25.1 Internal Representation	353
25.2 Representation Properties	356
25.3 Methods	356
25.3.1 Constructors	356
25.3.2 Trivial Accessors	357
25.3.3 Representation Accessors	357
25.3.4 Algorithmic Accessors	357
25.3.5 Representation Mutators	358
25.3.6 Content Mutators	361
25.3.7 Locator Initializers	366
25.4 Locator Inner Class	367
25.5 Performance Analysis	368
25.6 Quick Method Reference	370
26 Leftist Heap Data Structure	373
26.1 Internal Representation	373
26.2 Representation Properties	375
26.3 Leftist Heap Node Inner Class	376
26.4 Leftist Heap Methods	378
26.4.1 Constructors	378
26.4.2 Algorithmic Accessors	379
26.4.3 Content Mutators	382
26.4.4 Locator Initializers	391
26.5 Tracker Inner Class	391
26.6 Performance Analysis	393
26.7 Quick Method Reference	396
27 Pairing Heap Data Structure	399
27.1 Internal Representation	399
27.2 Representation Properties	402
27.3 Heap Node Inner Class	402
27.4 Pairing Heap Methods	405
27.4.1 Constructors and Factory Methods	406
27.4.2 Algorithmic Accessors	406
27.4.3 Representation Mutators	407
27.4.4 Content Mutators	408
27.4.5 Locator Initializers	414
27.5 Tracker Inner Class	415
27.6 Performance Analysis	418
27.7 Quick Method Reference	419
28 Fibonacci Heap Data Structure	423
28.1 Internal Representation	424
28.2 Representation Properties	426
28.3 Fibonacci Heap Node Inner Class	426
28.4 Fibonacci Heap Methods	428
28.4.1 Constructors and Factory Methods	428
28.4.2 Representation Mutators	429
28.4.3 Content Mutators	433
28.5 Performance Analysis	437
28.6 Quick Method Reference	441

29 Ordered Collection ADT	443
29.1 Case Study: Historical Event Collection (Range Queries)	443
29.2 Case Study: Linux Virtual Memory Map	444
29.3 Interface	445
29.4 Terminology	446
29.5 Competing ADTs	447
29.6 Selecting a Data Structure	447
29.7 Summary of Ordered Collection Data Structures	449
29.8 Further Reading	452
30 Sorted Array Data Structure	455
30.1 Internal Representation	455
30.2 Representation Properties	456
30.3 Methods	457
30.3.1 Constructors	457
30.3.2 Trivial Accessors	457
30.3.3 Binary Search Algorithm	458
30.3.4 Algorithmic Accessors	460
30.3.5 Content Mutators	464
30.3.6 Utilities for the B-Tree and B+-Tree Classes	465
30.3.7 Locator Initializers	467
30.4 Performance Analysis	467
30.5 Quick Method Reference	469
31 Abstract Search Tree Class	471
31.1 Internal Representation	471
31.2 Representation Properties	473
31.3 Abstract Tree Node Inner Class	474
31.4 Abstract Search Tree Class	475
31.5 Abstract Search Tree Methods	475
31.5.1 Constructors	475
31.5.2 Algorithmic Accessors	475
31.5.3 Content Mutators	479
31.6 Quick Method Reference	480
32 Binary Search Tree Data Structure	481
32.1 Internal Representation	481
32.2 Representation Properties	485
32.3 BSTNode Inner Class	485
32.4 Binary Search Tree Methods	489
32.4.1 Constructors and Factory Methods	490
32.4.2 Algorithmic Accessors	490
32.4.3 Content Mutators	496
32.4.4 Locator Initializers	500
32.5 Tracker Inner Class	501
32.6 Performance Analysis	504
32.7 Quick Method Reference	506
33 Balanced Binary Search Trees	509
33.1 Methods	510
33.2 Quick Method Reference	512

34	Red-Black Tree Data Structure	513
34.1	Internal Representation	514
34.2	Representation Properties	515
34.3	RBNode Inner Class	515
34.4	Methods	517
34.4.1	Constructors and Factory Methods	517
34.4.2	Content Mutators	517
34.5	Performance Analysis	528
34.6	Quick Method Reference	528
35	Splay Tree Data Structure	531
35.1	Internal Representation	531
35.2	Methods	532
35.2.1	Constructors	532
35.2.2	Representation Mutators	533
35.2.3	Algorithmic Accessors	533
35.2.4	Content Mutators	537
35.2.5	Locator Initializers	541
35.3	Performance Analysis	541
35.4	Quick Method Reference	543
36	B-Tree Data Structure	545
36.1	Internal Representation	546
36.2	Representation Properties	549
36.3	B-Tree Node Inner Class	549
36.3.1	B-Tree Node Methods	550
36.3.2	B-Tree Node Representation Mutators	551
36.3.3	B-Tree Node Content Mutators	555
36.4	B-Tree Methods	557
36.4.1	Constructors and Factory Methods	557
36.4.2	Algorithmic Accessors	558
36.4.3	Representation Mutators	562
36.4.4	Content Mutators	563
36.4.5	Locator Initializers	567
36.5	Marker Inner Class	568
36.6	Performance Analysis	570
36.7	Quick Method Reference	573
37	B+-Tree Data Structure	575
37.1	Case Study: A Web Search Engine	576
37.2	Internal Representation	577
37.3	Representation Properties	578
37.4	Leaf Node Inner Class	579
37.5	B+-Tree Methods	583
37.5.1	Constructors and Factory Methods	584
37.5.2	Representation Accessors	584
37.5.3	Algorithmic Accessors	585
37.5.4	Content Mutators	588
37.6	Performance Analysis	589
37.7	Quick Method Reference	591

38 Skip List Data Structure	593
38.1 Internal Representation	593
38.2 Representation Properties	596
38.3 Tower Inner Class	597
38.4 Skip List Methods	598
38.4.1 Constructors	598
38.4.2 Algorithmic Accessors	600
38.4.3 Representation Mutators	604
38.4.4 Content Mutators	605
38.4.5 Locator Initializers	610
38.5 Tracker Inner Class	610
38.6 Performance Analysis	613
38.7 Quick Method Reference	617
39 Digitized Ordered Collection ADT	619
39.1 Case Study: Packet Routing	619
39.2 Case Study: Inverted Index for Text Retrieval	620
39.3 Digitized Ordered Collection Interface	621
39.4 Selecting a Data Structure	622
39.5 Terminology	623
39.6 Competing ADTs	624
39.7 Summary of Digitized Ordered Collection Data Structures	625
39.8 Trie Variations	630
39.9 Suffix Trees	630
39.10 Indexing Tries	631
39.11 Further Reading	633
40 Trie Node Types	635
40.1 Trie Node Interface	635
40.2 Abstract Trie Node Class	635
40.3 Trie Leaf Node Interface	636
40.4 Abstract Trie Leaf Node Class	637
41 Trie Data Structure	639
41.1 Internal Representation	639
41.2 Representation Properties	642
41.3 Internal Node Inner Class	642
41.4 Leaf Node Inner Class	644
41.5 Search Data Inner Class	644
41.6 FindResult Enumerated Type	649
41.7 Trie Methods	650
41.7.1 Constructors and Factory Methods	650
41.7.2 Algorithmic Accessors	651
41.7.3 Content Mutators	657
41.7.4 Locator Initializers	661
41.8 Trie Tracker Inner Class	662
41.9 Performance Analysis	664
41.10 Quick Method Reference	667

42 Compact Trie Data Structure	671
42.1 Internal Representation	671
42.2 Representation Properties	672
42.3 Compact Trie Methods	672
42.3.1 Constructors and Factory Methods	673
42.3.2 Algorithmic Accessors	673
42.3.3 Content Mutators	675
42.4 Performance Analysis	679
42.5 Quick Method Reference	680
43 Compressed Trie Data Structure	683
43.1 Internal Representation	683
43.2 Representation Properties	685
43.3 Compressed Trie Node Interface	686
43.4 Internal Node Inner Class	686
43.5 Compressed Trie Leaf Node Inner Class	687
43.6 Compressed Trie Search Data Inner Class	687
43.7 Compressed Trie Methods	688
43.7.1 Constructors and Factory Methods	688
43.7.2 Algorithmic Accessors	689
43.7.3 Content Mutators	690
43.8 Performance Analysis	693
43.9 Quick Method Reference	694
44 Patricia Trie Data Structure	697
44.1 Internal Representation	697
44.2 Representation Properties	699
44.3 Patricia Trie Node Inner Class	700
44.4 Patricia Trie Search Data Inner Class	702
44.5 Patricia Trie Methods	705
44.5.1 Constructors	705
44.5.2 Content Mutators	706
44.6 Performance Analysis	717
44.7 Quick Method Reference	717
45 Ternary Search Trie Data Structure	719
45.1 Internal Representation	719
45.2 Representation Properties	721
45.3 Ternary Search Trie Internal Node Inner Class	722
45.4 Ternary Search Trie Search Data Inner Class	722
45.5 Ternary Search Trie Methods	723
45.5.1 Constructors and Factory Methods	723
45.5.2 Algorithmic Accessors	724
45.6 Performance Analysis	725
45.7 Quick Method Reference	725
46 Spatial Collection ADT	729
46.1 Case Study: Collision Detection in Video Games	729
46.2 Interface	730
46.3 Competing ADTs	731
46.4 Summary of Spatial Collection Data Structures	732
46.5 Further Reading	732

47 KD-Tree Data Structure	735
47.1 Internal Representation	736
47.2 Representation Properties	739
47.3 Alternating Comparator	739
47.4 KDNode Inner Class	742
47.5 KDTreeImpl Class	747
47.6 KD-Tree Methods	750
47.7 Performance Analysis	752
47.8 Quick Method Reference	754
48 Quad Tree Data Structure	757
48.1 Internal Representation	757
48.2 Representation Properties	760
48.3 Partitioning a Two-Dimensional Space	761
48.4 QTNode Inner Class	762
48.5 Box Inner Class	765
48.6 Quad Tree Methods	767
48.6.1 Constructors and Factory Methods	767
48.6.2 Representation Accessors	768
48.6.3 Algorithmic Accessors	768
48.6.4 Content Mutators	772
48.6.5 Locator Initializers	781
48.7 Performance Analysis	782
48.8 Quick Method Reference	782
49 Tagged Collection ADTs	785
49.1 Tagged Element	786
49.1.1 Mutable Tagged Element	787
49.1.2 Tagged Element Comparator	787
49.1.3 Tagged Element Digitizer	788
49.1.4 Tagged Element XY Comparator	788
49.2 Tagged Collection Interface	789
49.3 Tracked Tagged Interface	791
49.4 Competing ADTs	791
49.5 Selecting a Tagged Collection ADT	792
49.6 Tagged Collection Wrapper	793
49.7 Mapping ADT	797
49.7.1 Direct Addressing Mapping	799
49.7.2 Open Addressing Mapping	799
49.7.3 Separate Chaining Mapping	801
49.8 Tagged Priority Queue ADT	801
49.8.1 Tagged Priority Queue Wrapper	803
49.8.2 Tagged Binary Heap	806
49.8.3 Tagged Leftist Heap	806
49.8.4 Tagged Pairing Heap	807
49.8.5 Tagged Fibonacci Heap	807
49.9 Tagged Ordered Collection ADT	807
49.9.1 Tagged Ordered Collection Wrapper	811
49.9.2 Tagged Sorted Array	812
49.9.3 Tagged Binary Search Tree	813
49.9.4 Tagged Splay Tree	813

49.9.5	Tagged Red-Black Tree	813
49.9.6	Tagged B-Tree	813
49.9.7	Tagged B+-Tree	814
49.9.8	Tagged Skip List	814
49.10	Tagged Digitized Ordered Collection ADT	814
49.10.1	Tagged Digitized Ordered Collection Wrapper	818
49.10.2	Tagged Trie	818
49.10.3	Tagged Compact Trie	819
49.10.4	Tagged Compressed Trie	819
49.10.5	Tagged Patricia Trie	819
49.10.6	Tagged Ternary Search Trie	819
49.11	Tagged Spatial Collection ADT	820
49.11.1	Tagged Spatial Collection Wrapper	821
49.11.2	Tagged KD-Tree	822
49.11.3	Tagged Quad Tree	823
50	Tagged Bucket Collection ADTs	825
50.1	Case Study: Historical Event Collection (Indexing)	826
50.2	Case Study: Biosequence Comparison	828
50.3	Bucket Factory Interface	829
50.4	Tagged Bucket Collection Interface	829
50.5	Selecting a Tagged Bucket Collection ADT	831
50.6	Selecting a Data Structure	832
50.7	Tagged Bucket Collection Wrapper	832
III	GRAPH DATA STRUCTURES AND ALGORITHMS	839
51	Part III Organization	841
52	Graph ADT	843
52.1	Case Study: Task Scheduler	844
52.2	Terminology	845
52.3	Edge Interface	848
52.4	Graph Interface	848
52.5	Graph Representation Interface	850
52.6	Selecting a Data Structure	851
52.7	Summary of Graph Data Structures	854
53	Abstract Graph and Graph Algorithms	857
53.1	Representation Properties	858
53.2	Methods	858
53.3	In-Tree Representation	861
53.4	Finding Shortest Paths with Breadth-First Search	865
53.5	Finding Cycles and Connected Components with Depth-First Search	867
53.6	Topological Sort	873
53.7	Strongly Connected Components	875
53.8	Performance Analysis	878
53.9	Case Study: Garbage Collection	878
53.10	Further Reading	880
53.11	Quick Method Reference	881

54 Adjacency Matrix Data Structure	883
54.1 Internal Representation	883
54.2 Representation Properties	886
54.3 Methods	887
54.3.1 Constructors	887
54.3.2 Trivial Accessors	888
54.3.3 Algorithmic Accessors	888
54.3.4 Representation Mutators	889
54.3.5 Content Mutators	890
54.4 Edge Iterators	893
54.5 Incident Edge Iterator Inner Class	893
54.6 Adjacency Matrix Class	896
54.7 Performance Analysis	897
54.8 Quick Method Reference	898
55 Adjacency List Data Structure	901
55.1 Internal Representation	902
55.2 Representation Properties	905
55.3 Methods	906
55.3.1 Constructors	906
55.3.2 Trivial Accessors	906
55.3.3 Algorithmic Accessors	906
55.3.4 Content Mutators	908
55.4 Edge Iterators	910
55.5 Edge Iterator Inner Class	910
55.6 Adjacency List Class	912
55.7 Performance Analysis	912
55.8 Quick Method Reference	915
56 Weighted Graph ADT	917
56.1 Case Study: Airline Travel Agent (Revisited)	917
56.2 Case Study: Image Segmentation	919
56.3 Terminology	921
56.4 Weighted Edge Interface	922
56.5 Simple Weighted Edge Class	922
56.6 Weighted Graph Interface	923
56.7 Selecting a Data Structure	923
56.8 Weighted Adjacency Matrix Data Structure	924
56.9 Weighted Adjacency List Data Structure	924
57 Abstract Weighted Graph and Weighted Graph Algorithms	925
57.1 Greedy Tree Builder	926
57.2 Dijkstra's Single-Source Shortest Path Algorithm	931
57.3 Prim's Minimum Spanning Tree Algorithm	935
57.4 Kruskal's Minimum Spanning Tree Algorithm	938
57.5 Bellman-Ford's Single-Source Shortest Path Algorithm	941
57.6 Shortest Path Matrix	943
57.7 Floyd-Warshall's All-Pairs Shortest Path Algorithm	945
57.8 Edmonds-Karp Maximum Flow Algorithm	948
57.9 Further Reading	954
57.10 Quick Method Reference	955

IV	APPENDICES	957
A	Java Fundamentals	959
A.1	Types and Operators	959
A.2	Packages	961
A.3	Classes and Objects	961
A.3.1	Instance Variables	962
A.3.2	Methods	962
A.3.3	Constructors	965
A.4	The Heap	965
A.5	Exception Handling	967
A.6	Assertions	969
A.7	Inner Classes	969
A.8	Static Members	970
A.9	Interfaces	971
A.10	The Class Hierarchy	973
A.10.1	Extending Classes	973
A.10.2	Type Checking and Casting	973
A.10.3	Superconstructors	974
A.10.4	Overriding Methods	974
A.10.5	Polymorphism	974
A.10.6	Iterators	975
A.11	Generics	976
B	Complexity Analysis	979
B.1	Time Complexity	980
B.2	Asymptotic Notation	981
B.3	Space Complexity	984
B.4	Expected Time Complexity	985
B.5	Amortized Analysis	985
B.6	Solving Recurrence Equations with the Master Method	987
C	Design Patterns Illustrated in This Book	991
C.1	Abstract Factory	991
C.2	Adaptor	991
C.3	Bridge	992
C.4	Builder	992
C.5	Command	992
C.6	Composite	993
C.7	Decorator	993
C.8	Facade	993
C.9	Factory Method	994
C.10	Flyweight	994
C.11	Iterator	994
C.12	Leasing	995
C.13	Proxy	995
C.14	Singleton	995
C.15	Strategy	996
C.16	Template Method	996
C.17	Visitor	996
	References	997

Preface

This handbook of data structures and algorithms is designed as a comprehensive resource for computer science students and practitioners. The book is, quite literally, the product of a marriage of theory and practice. As an alternative to the survey approach taken by traditional data structures and algorithms textbooks, this book builds on a theoretical foundation to offer a top-down application-centered approach and a systematic treatment of data structure design and their practical implementation.

The book serves three major purposes: guidance, implementation, and insight. Charts, decision trees, and text provide guidance through the large body of material presented. Unlike a textbook, it is not necessary to read the entire book to fully benefit from its contents. Our intention is that readers with a specific problem will follow the provided guidance and organizational tools to quickly identify the most appropriate data structure or algorithm for their problem. For example, readers seeking a data structure for an application are first guided to a suitable abstract data type (ADT), and then to the most appropriate implementation of that ADT. Trade-offs between competing data types and implementations motivate each decision in the context of the problem at hand.

Traditional textbooks generally gloss over the different possible variations of a given data structure type. For example, a typical textbook has a chapter on “hashing” that treats all of the various uses of hashing uniformly as one idea (for example, hash-based implementations of a set or mapping). However, in reality, implementing them all in terms of a single ADT would lead to inefficiencies for alternate uses. Consider an application that requires a mapping from each word in a text document to the positions at which it occurs. One could use Java’s *HashMap* to associate each word with a linked list of line numbers. However, each insertion to associate a new word with a line number would require using *get* (to discover that the word is not yet in the mapping), and then *put* (that duplicates most of the work performed by *get*). In this book, we explicitly include the *BucketMapping* interface to provide efficient support for such an application. By explicitly introducing separate interfaces and ADTs for important variations in usage, differences can be highlighted and understood.

The book includes complete implementations for a wide variety of important data structures and algorithms. Unlike most textbooks that sweep details under the rug to simplify the implementation for “ease of explanation,” we have taken the approach of providing complete object-oriented implementations within an extensible class hierarchy. Yet we have not done so at the expense of clarity. Because of the completeness of implementation, chapters on some topics are longer than one might see in a textbook covering a similar topic. However, the organization of the chapters simplifies navigation, and the detailed implementations provide design insights useful to practitioners. Our implementations follow standard Java programming conventions.

Parts II and III of the book cover a large number of data structures and algorithms. We include many abstract data types not provided in the standard Java libraries, but for those data types that are also present in the Java Collections classes, we have tried to remain consistent with the Java interfaces and semantics wherever possible. However, we have diverged in places where our design goals differ. One important departure from the Java Collections is our separation of the iterator concept into two types: markers and trackers. Unlike Java’s provided iterator implementations, markers and trackers support concurrent modification of data structures. In addition, the introduction of a tracker, which maintains the location of a particular object even if its location changes within the structure, is crucial for efficient implementations of even some standard algorithms, such as the use of a priority queue to implement Dijkstra’s shortest path algorithm. However, care must be taken in

many data structure implementations to efficiently support tracking, and our presentation includes a discussion of such design choices.

We integrate the presentation of algorithms with the ADTs that support them. In many cases the algorithms are implemented in terms of the ADT interface and included in an abstract implementation of the ADT. The advantage of such an approach is that the algorithm (in both its presentation and instantiation) is decoupled from the particular ADT implementation.

As thorough as we have tried to be, it would not be possible to cover all possible variations of each data structure. Therefore, explanations of each implementation are designed not only to assist readers in understanding the given implementations of data structures and algorithms, but also to support readers in customizing implementations to suit the requirements of particular applications. Making such modifications while preserving correctness and efficiency requires an understanding of not only how the code operates, but why the code is correct and what aspects of the implementation contribute to its efficiency. To this end, we have provided clearly identified explanations of correctness properties for each implementation, as well as correctness highlights that explain how each method depends upon and preserves these properties. For data structures, these properties often relate to an abstraction function that captures, without undue formalism, how the organization of the data structure maps to the user view of the abstraction. This aids understanding at the intuitive level and serves as a foundation for the methodology we use to reason about program correctness. In this way, if readers choose to modify the provided code, they will be able to check that their change preserves the correctness of the implementation. Similarly, we provide a clearly identified section in which time complexity analysis is provided for each data structure and algorithm. Readers interested in modifying a particular method can look in that section to understand how that method (and consequently their proposed change) influences the overall performance of the implementation. Space complexity issues are also discussed.

The format of the book is designed for easy reference. In Parts II and III, each major data type and its implementations are presented in a sequence of chapters beginning with the semantics of that data type, and followed by each implementation. Within each chapter, standardized section headings help the reader quickly locate the required information. The stylized format is designed to help readers with different needs find what they want to read, as well as what they want to skip. A reader planning to use, but not modify, a data structure implementation may decide to read the introductory explanations and then skim through the implementation while omitting the correctness highlights, and finally read the time complexity analysis at the end of the chapter.

The case studies presented throughout the book provide further examples of how various data structures and algorithms presented in Parts II and III can be applied. They also exemplify the process by which those particular data structures and algorithms were selected for the application. One appendix provides a brief overview of the major features of the Java programming language, another appendix reviews asymptotic notation and complexity analysis, and a third appendix briefly discusses the design patterns we illustrate in this book. Source code for the data types, including interfaces, implementations, algorithms, and sample test cases, is included on the accompanying CD. Complete documentation, in Javadoc format, is also provided.

Note to instructors: An introductory data structures and algorithms course could begin with Part I, with an emphasis on selecting abstract data types and implementations appropriate for applications. Then, based on the interests of the instructor and students, a selected subset of the ADTs could be covered in detail. It is not necessary to present every data structure for each ADT, but instead the comparison tables can be used to highlight the differences, and then students can concentrate on one or two representative implementations of each. For courses with a more applied focus, homework and projects might concentrate on empirical comparisons of the provided implementations, modifications based on the optimizations suggested in the chapters, and projects based on the case studies. A more theoretical course might cover the complexity analysis material in the appendix early in the course, and focus more on theoretical analysis of the algorithms and correctness proofs. Online educational materials will be made available at <http://goldman.cse.wustl.edu> in December 2007.

Acknowledgments

Successful completion of a large project like this is possible only in the context of an understanding family, engaging colleagues, and a supportive community. First and foremost, we thank our children, Mark, Ben, and Julie, for relinquishing our time for many long hours while we worked on this book. We thank our parents, Robert and Marilyn, and Lester and Judy, for all that they have done to support the busy schedules of our children and ourselves. We thank our colleagues Jeremy Buhler for guidance on the case study on biosequence comparison, Chris Gill for helpful discussions about design patterns, Patrick Crowley and Fred Kuhns for guidance on the case study on Linux virtual memory organization, and Tao Ju and Robert Pless for helpful discussion on spatial collection data structures. We thank two exceptional undergraduate students, Ben Birnbaum and Troy Ruths, who carefully read an early draft of large sections of the manuscript and provided valuable advice that helped shape the final form of the book.

The philosophy and methodology of this book are largely products of the combined influence of our former research advisors and teachers, most notably Nancy Lynch, Ron Rivest, Barbara Liskov, Paris Kanellakis, Stan Zdonick, Andy Van Dam, Jeff Vitter, Tom Miller, and many other dedicated educators. We thank them deeply.

We thank Tom Cormen, Charles Leiserson, Ron Rivest, and Cliff Stein for graciously sharing their bibliography database, which was checked by Denise Sergent and Ayorkor Mills-Tetley.

We acknowledge the professional staff at CRC Press and Taylor & Francis Group, LLC, including Sunil Nair, Tom Skipp, Ari Silver, Jill Jurgensen, Katy Smith, Grace Bishop, Kevin Craig, Theresa Del Forn, and Clare Brannigan. We thank Shashi Kumar, Nishith Arora, George Shibu, and Sanjay Kandwal for their support with the formatting. This book was formatted using $\text{\TeX}$, $\text{\LaTeX}$, and $\text{\pdf\LaTeX}$. Code segments were formatted with $\text{\java2latex}$.

We thank the National Science Foundation for supporting our research during this period under grants IDM-0329241, CCR-9988314, CISE-EI-0305954, and CISE-CCLI-0618266.

Authors

Sally Goldman earned an Sc.B. from Brown University in 1984. She completed her Ph.D. under the supervision of Dr. Ronald Rivest at MIT in 1990. Since 1990, she has been a professor in the Department of Computer Science and Engineering at Washington University. She is the Edwin H. Murty Professor of Engineering and the Associate Chair of the Department of Computer Science and Engineering. She regularly teaches an introductory data structures and algorithms course as well as an advanced algorithms course. She has many journal and conference publications in computational learning theory, machine learning, image retrieval and on-line algorithms.

Kenneth Goldman earned an Sc.B. from Brown University in 1984. He completed his Ph.D. under the supervision of Dr. Nancy Lynch at MIT in 1990. Since 1990, he has been a professor in the Department of Computer Science and Engineering at Washington University. He regularly teaches introductory computer science and design methodology courses. His journal and conference publications are primarily in the area of distributed systems, distributed algorithms, and programming environments.

Part I

INTRODUCTION

Chapter 1

Design Principles

How to organize information is one of the first and most important decisions a software developer faces. This decision affects application development in two fundamental ways, **conceptual** and **structural**. The way we conceptualize information profoundly affects the way we think about that information in order to solve problems. For a given application, thinking about the information one way may lead to a natural, easy-to-implement solution, while thinking about the same information another way may lead to an awkward, difficult solution. Consequently, careful decisions about the conceptual organization of information reduce total implementation time and increase both the elegance and extensibility of the resulting software. The act of conceptualizing information is known as **data abstraction** and results in the formation of **abstract data types (ADTs)** that capture the way we think about information. An ADT consists of (1) a set of **operations** (methods) that every implementation of that data type is expected to provide, and (2) a set of **legal behaviors** that capture how we expect implementation to behave when we invoke its operations. Together, these form a **specification** for a correct implementation of the ADT. For example, suppose an ADT includes, among others, the operations *getSize* and *removeFirst*. A specification might capture the expectation that the return value of *getSize* after a successful call to the *removeFirst* method should be one less than the return value of *getSize* immediately before the remove. It is critically important to notice that an ADT does *not* specify how the implementation should store the information or what steps it should take to access or modify the information. The ADT constrains the implementation only to the extent that a correct implementation must provide the required operations and exhibit only legal behaviors.

The first step in the design process is settling on an appropriate ADT (or ADTs) for an application's information. Because an ADT provides a well-defined interface and semantics, it is possible to design, and even implement, the entire application in terms of an ADT, postponing the decision about how to best implement the ADT itself. Ultimately, however, an actual implementation of that ADT is required. When implementing an ADT, it becomes necessary to consider how the information will be laid out within the computer's memory. The way we structurally organize information in memory largely determines the efficiency with which different operations on the information are carried out. This structural organization is known as a **data structure**, and the ways in which the operations are carried out are known as **algorithms**. One data structure may support fast algorithms for some operations and slow execution for others, while another may perform exactly opposite. A simple example of such a trade-off is the decision of whether or not to keep data sorted in memory. Keeping the data sorted can improve search time at the expense of slower insertion, whereas keeping the data unsorted may result in fast insertion at the expense of a slower search. In general, one must choose data structures carefully on the basis of which operations the application will use most frequently.

Sifting through numerous competing alternatives for organizing information can be overwhelming. Consequently, it is easy to make mistakes that lead to awkward and/or inefficient systems. One of the biggest, and most frequent, mistakes is to confuse the two separate questions of conceptual and structural organization. In fact, sometime developers overlook conceptual organization and think only about the data structure. These developers miss the liberating opportunity to carry

out design and implementation in terms of data relationships natural for the application. Instead, they are stuck in the realm of low-level language primitives, which become the de facto conceptual representation. The second common mistake is to choose an inappropriate conceptual organization, rendering the rest of the software development process awkward and frustrating. This often happens because a developer is unaware of viable alternatives, or because it is human nature to gravitate to familiar approaches. Finally, even if an appropriate conceptual organization is chosen, it is easy to make the mistake of choosing an implementation that does not support the application efficiently. In such cases, the design and implementation may proceed smoothly, but the resulting application may run too slowly or require too much memory.

For example, suppose an application needs an ADT that must only support a method to add a new comparable object and a method to return the minimum-valued object that was added. While one might consider a variety of data structures that store all of the objects added, the most efficient implementation will just store the minimum object added so far. The conceptual organization must drive the structure, not the reverse.

This book is designed to assist software developers through the process of deciding how to organize information, both conceptually and structurally. The book provides a two-stage methodology for making these decisions, beginning with the conceptual organization of data using ADTs and then moving to the structural organization using data structures and algorithms. ADT selection is the first stage of the decision process because it has far-reaching implications throughout application design and development. The second stage, selection of an appropriate data structures that implement the ADTs, is focused on selecting implementations that are most efficient for the particular characteristics of the application. Examples and case studies throughout the book illustrate both stages of the decision-making process.

All implementation in this book is carried out in Java, a strongly typed object-oriented programming language. We assume familiarity with the major features of Java summarized in Appendix A including exception handling, inner classes, iterators, polymorphism, and generics. We also assume familiarity with standard time complexity analysis as summarized in Appendix B, including asymptotic growth rates, expected time complexity, and amortized time complexity.

The remainder of this chapter describes how we will use Java to support the methodology advocated by this book, to promote a philosophy of design that strives to separate the concerns of data abstraction from implementation, as well as to simplify careful reasoning about data structure implementations. We begin, in Section 1.1, with an overview of our approach to object-oriented design. Section 1.2 describes mechanisms for, and benefits of, encapsulating data inside of objects. Section 1.3 discusses how encapsulation can simplify the process of arguing correctness through the use of representation invariants and correctness properties. In Section 1.4, we describe Java interfaces as a way to support the creation of ADTs. Section 1.5 includes a small case study to illustrate how the conceptual design can allow us to think about information purely in terms of how we want to use it, without worrying (at least initially) about how those ways are supported by a physical implementation. Finally, in Section 1.6, we close with a case study on tree representations that focuses on structural design.

1.1 Object-Oriented Design and This Book

Like other object-oriented languages, Java encourages software developers to organize applications into modules (**packages** and **classes**) that encapsulate data and provide functionality to users through methods that together comprise a narrow **application programmer interface (API)**. Narrow interfaces simplify design because they enforce a disciplined use of data, eliminate unnecessary

dependencies, and provide a clean abstraction that supports using data types in terms of their specifications. Object-oriented languages also encourage significant software reuse, particularly through class hierarchies that enable specialized classes to be created by specifying only the ways in which they differ from the classes they extend.

Beyond describing individual data structures and algorithms, this book leverages object-oriented design techniques to provide a unified treatment of these data structures and algorithms. For example, the class hierarchy is used to expose natural relationships among data structures and focus on the places where more specialized data structures differ from their more general supertypes. The resulting unified class library provides a scaffolding for understanding the practical use of data structures and algorithms, as well as a foundation for reasoning about the correctness and performance of data structures and algorithms in terms of well-defined specifications. In addition, Appendix C discusses the design patterns illustrated throughout the book. Because of this treatment of the material, readers can approach this book to learn about individual data structures and algorithms and their relationships, as well as to learn about how object-oriented design techniques can be effectively applied to construct a framework that supports a wide variety of applications.

1.2 Encapsulation

Objects combine data and methods. The internal representation (**instance variables**) of objects provide a mechanism for consolidating information about an entity in one place. **Methods**, the sequence of statements to perform an operation, provide ways to access and operate on that information.

If we look at objects from the outside, without knowing their internal representations, we see abstract entities that are created by calling a constructor and on which we can perform certain operations by calling methods. Keeping the internal representation hidden from the users of an object is called **encapsulation** and is an important part of object-oriented design. Encapsulation serves three important purposes. It enables users of objects to work entirely in terms of the abstraction provided by those objects, without worrying about how the data is represented internally, or how the methods carry out their operations. Consequently, design and implementation of applications can be carried out at a higher level of abstraction. The second important purpose is that it enables the implementer freedom to change the internal representation and method implementations of a class without breaking application programs that use its objects. In fact, when an implementation is properly encapsulated, the internal implementation can be completely changed to improve performance without a single change to the code in the rest of the application. This is important because there are trade-offs between implementations in which some methods may be faster, while others may be slower, and often it is not known in advance where performance bottlenecks in a system will be. Rapid prototyping followed by selective replacement of implementation for performance improvements is often a sensible approach. Finally, encapsulating the internal representation prevents accidental misuse of the data, a common source of program errors.

Enforcing Encapsulation

To enforce encapsulation, Java provides mechanisms for controlling the visibility of both instance variables and methods. Every instance variable and method has one of four possible levels of visibility: *private*, *protected*, *package*, or *public*. All except package level visibility (which is the default protection level) have an associated keyword that is used as a modifier preceding the declaration

of the instance variable or method. For example, the following code declares a private instance variable, a protected method to set its value, a package visible method to increment its value, and a public method to get its value.

```
private int count = 0;
protected void setCount(int count) { this.count = count; }
void increment() { count++; }
public int getCount() { return count; }
```

The most restrictive visibility is **private**, which means that only code within the class being defined can use the variable or method. The least restrictive is **public**, which means that code in any class can use it. In between, there are two other visibility levels. The default visibility, known as **package level visibility** allows code within the package to use the variable or method. Finally, the **protected** modifier allows code within other classes of that package, plus code in any of its subclasses outside the package, to use the variable or method.

Declaring instance variables as private helps enforce encapsulation, since no code outside the class can see or manipulate that representation without calling an externally visible method. However, carefully choosing the access modifiers is not entirely sufficient to ensure encapsulation. It is also important to control the return values of visible method so that they do not expose the representation by returning references to objects that should not be seen or modified externally.

Although not strictly considered an access modifier, the **final** modifier is convenient for exercising further control over the way in which a variable or method is used. The *final* modifier, which can be used along with any of the access modifiers, specifies that something cannot be changed once defined. For example, we can declare an instance variable as *final* to specify that it can be initialized either when it is declared, or by the constructor, but nowhere else. The compiler checks this, so arbitrary modification of the variable is prevented. When applied to methods, the *final* modifier means that subclasses cannot override the method. For example, making a method both final and protected is useful when subclasses should be able to call the method, but replacing its implementation should be prevented.

1.3 Invariants and Representation Properties

When an object encapsulates its internal representation, the methods of that object form an **abstraction barrier** that entirely controls the use and modification of that object. The data inside the object can be modified only by calling those methods. Therefore, it is possible to reason about the **correctness** of the object (or its class) in isolation, without thinking about the rest of the application. Reasoning about the correctness of each class separately is much easier than reasoning about the correctness of the entire system at once.

In reasoning about the correctness of an implementation, it is important to know that the data inside the object is always consistent. For example, in an object that encapsulates a list, we may want to ensure that an integer instance variable named *size* always contains the number of data elements in the list. A property that is always true is known as an **invariant**, and an invariant that concerns the state of an object is known as a **representation invariant**. If data of our hypothetical list object were not encapsulated, we would have no way of reasoning about our representation invariant in isolation, because any part of the application could modify the *size* variable arbitrarily. However, if the internal representation is encapsulated, we can reason about each constructor and mutating method independently using the following standard technique. To prove a representation invariant,

1. verify that each constructor *establishes* the truth of the invariant, and
2. verify that each externally visible method *preserves* the truth of the invariant.

The first requirement says that each constructor must initialize the object in a consistent state. The second requirement says that *if* an object is in a consistent state, then when any method executes it will leave the object in a (possibly different) consistent state. If the object begins in a consistent state, and if each method moves the object from a consistent state to a consistent state, then by mathematical induction the object will always be left in a consistent state.

Note that this methodology supports reasoning about each constructor and each method separately, rather than all at once. Furthermore, the proof obligation for each method is quite limited. It is not necessary to show that a method *makes* an invariant true, but only that the method *preserves* the invariant. That is, a proof that an invariant holds on method termination can assume the invariant holds on method invocation. An important benefit of this is that simpler reasoning leads to a simpler implementation. In particular, the fact that all representation invariants are true upon entering a method means there is no need to clutter the code with consistency checks or code fragments that fix up the state if it becomes inconsistent. However, it is still necessary to write code to verify that incoming parameter values satisfy the necessary requirements, since there is no other means to control what values a user (or other external system component) might provide.

The correctness of a data structure ultimately depends upon the user's abstract view. That is, our obligation is to guarantee that the data structure correctly implements the intended abstraction. However, the user's abstract view is not necessarily explicitly represented as part of the state of the object. Consequently, for each data structure presented in this book, we define an **abstraction function**, a function that maps from the internal representation to the user's view of the ADT instance. So, although the user's view is not explicitly represented, the abstraction function allows us to reason about it as if it were part of the state.

We use the term **representation property** to refer to either a representation invariant or a relationship between the internal state of the object and the abstraction function. Using these properties, we can reason simultaneously about the user's view and the internal state of the object, supported by the methodology outlined above. For a simple example of an abstraction function and related representation properties, see the array data structure (Chapter 11).

1.4 Interfaces and Data Abstraction

We began this chapter with a discussion of the importance of abstract data types (ADTs) as a mechanism for specifying the operations that can be performed on a data type, and how those operations are expected to behave. The purpose of an ADT is to simplify the application design process by allowing the application programmer to think about information purely in terms of how the ADT is to be used, without worrying (at least initially) about how the operations are supported by a particular implementation. In fact, it should not be necessary for an implementation to exist when making conceptual decisions about how information will be used. However, reasoning about something that is purely abstract can be difficult unless there is a precise way to describe it.

The most obvious way to describe something abstract is to create a concrete realization, but this is precisely what we wish to avoid, for it would blur the separation between conceptual and structural design. Consequently, to isolate the conceptual design from the structural design, a mechanism is needed to describe each ADT in its own right.

Defining an ADT in Java involves creating a Java **interface** that lists the methods of the ADT and provides documentation that describes the allowable behaviors. For example, the interface for an ADT to maintain a set of elements might have methods to add and remove elements and a

contains(x) method to check whether a particular object *x* is an element of the set. The allowable behaviors might require that *contains(x)* returns true if and only if the prior method calls on the object contain a call to *add(x)* without a subsequent call to *remove(x)*. Such an interface could be written as follows.

```
public interface Set {
    public void add(Object x);           //adds object x to the set
    public void remove(Object x);       //removes object x from the set
    public boolean contains(Object x);  //true when object x is in the set
    public boolean isEmpty();           //true when the set is empty
}
```

By providing various internal representations and method implementations, any number of different classes can be defined that implement the data type associated with an ADT, possibly with different performance properties. The abstraction sets up a contact between the user and the implementer. Java requires that any class that implements an interface must provide implementations for each method in that interface. Moreover, there is the expectation that each implementation of an ADT exhibit only legal behaviors as specified by the ADT documentation.

Recall that an interface defines a type, just as a class defines a type. Consequently, users of an ADT can write application code entirely in terms of the abstract type. For example, an application program may declare a variable of type *Set* and call any of the *Set* methods on that variable. When the time comes to choose a particular implementation, any class that implements that interface may be used. Furthermore, this choice may be changed later (to switch to a more efficient *Set* implementation, for example) without modification of the application code.

1.5 Case Study on Conceptual Design: Historical Event Collection

We now use a simple example to illustrate how changes in the application requirements can impact the conceptual design. The central purpose of this case study is to illustrate how evolving application requirements contribute to the decisions one makes in selecting an appropriate ADT. The discussion refers to several specific data types and makes use of the decision process explained in Chapter 2. In addition, Sections 29.1 and 50.1 discuss this case study in more depth including decisions about the structural design.

Consider an application to maintain a collection of historical event objects where each event object includes a date, a description, and possibly other information about the event. We start with a very simple application, in which the only required operations are:

- Insert a new event into the collection.
- Given a date, return a list of all events in the collection that occurred on that date.

Observe that the event description is not used by either of these operations. So in defining the equivalence between events, only the date need be used. Also, although there is well-defined ordering between dates, neither of the required operations make use of this ordering. Finally, once an event is placed in the collection, it does not need to be removed. Because of the limited way in which the events are accessed by the application, the conceptual design can focus on ADTs that provide similar methods. Using the framework presented in Chapter 2, we find that the BucketMapping ADT (Chapter 50) provides the best conceptual design for an application that must support only the methods described above. While one could imagine wanting to create a very general purpose

implementation to support operations that are not required, doing so can substantially increase the expense of supporting the needed operations, both in implementation time and execution time.

Next, suppose the application requirements change to include the following additional operations:

- Given a date, remove all events with that date.
- Given a date and description, remove all events with the given date and description.

In addition, suppose this application also requires that the insert operation preserves the invariant that all events are distinct, meaning no two events in the collection have the same date and description. Even with the addition of these two operations, still no operation requires using the natural ordering we associate with dates. The ADT need only support methods that depend on the equivalence of dates. So a BucketMapping remains the best ADT choice for conceptualizing how events are organized with respect to the date.

In this application, the conceptual design is not changed by the addition of an operation to remove elements. Nevertheless, it is important to keep in mind that for some applications an ADT that does not include methods to remove objects may be preferable over one that does (e.g., the example discussed on page 4). Also even when the ADT includes a method to remove elements, the data structure design can benefit from knowing that remove is not needed since one can select a data structure in which the desired operations are more efficient at the cost of remove not being supported as efficiently. So while the conceptual design may not be changed by requiring these two methods, the structural design will change.

However, changes that do affect the conceptual design are (1) the requirement placed on insert to ensure that all events in the collection are distinct, and (2) the operation to remove an event with a given date and description. Both of these methods consider the event description. To complete the conceptual design, more information about the application is needed. In particular, how many events are expected to share the same date? If this number is small, then the list of events with a given date can be processed using a linear time method that checks for a matching description. However, suppose a large number of events could share the same date. Then a secondary ADT is needed to organize all of the events with the same date to support finding one (if it exists) with the given description. Since the descriptions associated with an event are unique, the Set ADT (Chapter 20) provides the needed functionality. Note that, even in this simple example, the distinction between conceptual and structural choice can become somewhat blurry, because a second ADT may be warranted to support more efficient performance in the use of the first. Consequently, it is always important to be vigilant about distinguishing these two aspects of design choices and understand how they relate.

As a next step, suppose the application must also support the following operation:

- Given a date d and a range r , return a list of all events in the collection that occurred in the interval $[d - r, d + r]$.

The addition of this operation affects the conceptual design since this method depends upon the natural ordering associated with dates. While this may not seem a major change, the data structures used by the Set, Mapping, and BucketMapping ADTs are all based on the use of **hash tables** which provide extremely efficient support to locate an equivalent element, but do not support finding all elements in a given range. Instead the OrderedCollection ADT (Chapter 29) is most appropriate. In order to obtain the best implementation for such an application, it is crucial that the conceptual design reflect the need to have methods that depend on the ordering that exists among dates.

Finally, we consider the addition of one more operation.

- Given a word, return a list of all events in the collection that include the given word in the event description.

Earlier we discussed introducing a secondary data structure to organize all events with a single date. However, up to this point there was no need for an ADT organized based on any aspect of the description that applied to all events. To support this new operation, we may require an ADT that can take any word that has occurred in any description and associate with it a collection of event objects that contain that word in their descriptions. A BucketMapping ADT, this time organized using the words in the event descriptions, is a good fit.

Observe that when all of the operations described in the section are included, and when there may be many events with the same date, we used three different ADT instances: a bucket mapping to efficiently find all events that include a given word in their description, an ordered collection to organize the events based on the date, and a set associated with each date, to efficiently locate an event (if any) having a given description.

Section 29.1 discusses the ADT and data structure selection to perform the operations that depend upon the date, and Section 50.1 discusses the ADT and data structure selection to perform the operations that depend upon the words in the event descriptions.

1.6 Case Study on Structural Design: Trees

The previous case study focused on conceptual design. In this section, we look at the process of structural design. Structural design decisions often involve making trade-offs between the time and space complexity of the solution, known as **time-space trade-offs**.

At a fundamental level, data structures are organized using two language primitives: **references** (or pointers*) and **contiguous memory** (or arrays). For any structural design both pointer-based and array-based representations should be considered, so we briefly highlight the general trade-offs between these two kinds of representations.

In a **pointer-based data structure** portions of the data are kept in separate objects, and the organization is determined by the way these objects refer to each other. Access is provided by following references. In pointer-based data structures, the elements in the data structure are not contiguous in memory. Each element is allocated to an arbitrary free block of memory in the heap (of the appropriate size) and pointers to other objects in the data structure allow navigation. Because the objects need not be contiguous in memory, new elements can be added to the structure dynamically. It is not necessary to know *a priori* how many elements will be placed into the structure. The space usage of the pointer-based structure is determined by the sizes of objects within the data structure, including the pointers themselves.

In an **array-based data structure**, the data is stored within an **array** which is a contiguous memory block. Array-based data structures have many advantages as well as some disadvantages over pointer-based structures. The most important advantage of array-based structures is that access to the i^{th} element in the array can be performed in constant time. Also, arrays generally use less space since no variables are needed to support navigation. However, since an array is a contiguous block of storage, its size is fixed when the array is allocated. If the array is too small, it cannot hold as many elements as needed, but if the array is too large then considerable space can be wasted. A **dynamic array** (Chapter 13) can address this issue, but there is still overhead in space (for unoccupied array slots) and time (to copy values from a full array to a new larger one). Another disadvantage of array-based data structures is that in order to remove or add an element in the middle of an array, all the elements after (or before) the given element must be shifted to the left (or right),

*A reference is a pointer that cannot be mathematically manipulated. A reference is the only type of pointer supported within Java.

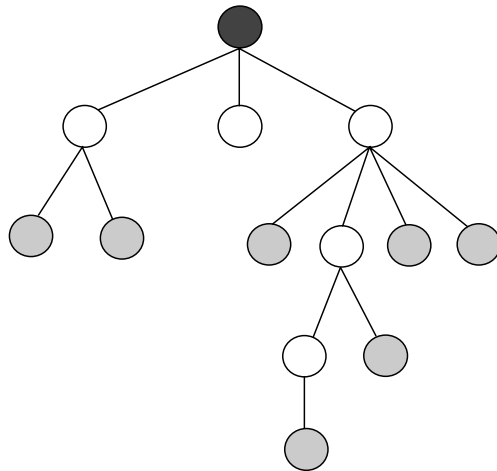


Figure 1.1

An example rooted tree. The root is shaded black, the other internal nodes are unshaded, and the leaves are shaded gray.

which takes linear time. Whenever possible it is best to use an array-based data structure when the ADT only requires adding or removing items near the ends of the sequence.

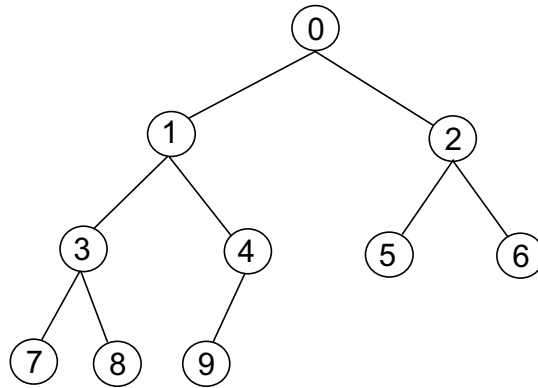
This case study explores the design issues one might consider during the process of selecting a data structure to represent the abstract concept of a **tree**. One node might be a designated root. A tree without a root is an **unrooted tree**. A collection of trees (either rooted or unrooted) is called a **forest**. Each tree is composed of elements that we call **nodes**. In this section we focus on **rooted trees**. Each node of a tree (except the root) has a parent and some number of children (possibly zero). A **leaf** is a node with no children. A sample tree is shown in Figure 1.1. One example of a rooted tree is an organizational chart in which the CEO is the root of the tree. The parent-child relationship is created by having each employee be the child of his/her boss. A phylogenetic tree is another natural example of a rooted tree. Also many data structures for ADTs that are not naturally viewed as a tree, such as `PriorityQueue` (Chapter 24), `OrderedCollection` (Chapter 29), and `DigitizedOrderedCollection` (Chapter 39), use a tree to provide the needed structure.

To create the most appropriate implementation for a tree, there are several important questions to consider. For example,

- Is it important to have constant time access to the parent of a node?
- Is it important to have constant time access to the children of a node?
- Is it important to have constant time access to the siblings of a node?
- How important is it to minimize space usage?

The ADT selected during conceptual design can be used to help answer these questions. Since every node in the tree (except the root) has a single parent, if there is a need for constant time access to the parent, then a reference to the parent is included as an instance variable of each node. The choice of whether or not to include a *parent* reference is orthogonal to the other decisions that relate to the number of children that each node may have, and how they can be accessed.

We start by considering the structural design for a **binary tree**, in which each node has at most two children (typically called the **left child** and the **right child**). Some examples of binary trees that appear later in this book are the binary heap (Chapter 25), the leftist heap (Chapter 26), the binary

**Figure 1.2**

A binary tree in which nodes 0, 1, ..., 9 are added level-by-level from left to right.

search tree (Chapter 32), the k -d tree (Chapter 47), and the Patricia trie (Chapter 44). The most common structural design for a binary tree includes references to an associated data object, a left child, a right child, and a parent for each node. Additional fields vary. While such an implementation supports constant time access to the children, sibling, and parent of each node, the space requirement for such a data structure is $4n$ references where n is the number of elements since there are four instance variables for each node[†].

We now consider the somewhat special situation in which the structure of the tree is very constrained. In particular, suppose that the nodes are added to the binary tree level-by-level from left to right. For example in the binary tree of Figure 1.2, suppose that a tree with n items structurally includes just the nodes labeled by 0, 1, ..., $n - 1$.

Because of this specialized structure, an array a could be used to support constant time access to the sibling, children, and parent using only one reference per element in the tree. The element associated with node i is referenced by $a[i]$. Observe that the left child of node i is node $2i + 1$, and the right child of node i is node $2i + 2$. Furthermore, if $2i + 1 = n - 1$ then node i only has a left child, and if $2i + 1 \geq n$ then node i is a leaf. Thus, one can compute the location of either child of node i , in constant time. For all nodes except for the root ($a[0]$), the parent of node i is node $\lfloor (i - 1)/2 \rfloor$, so that can be found in constant time as well. Similarly, the sibling of any node can be found in constant time.

Since elements cannot be efficiently added or removed from the middle portions of the array, an array-based representation for a tree is appropriate when the tree has the structure shown in Figure 1.2, and also when the ADT methods can be implemented in a way where elements are only added or removed from the front or back of the array. The binary heap (Chapter 25) is an example of a data structure for which an array-based representation can greatly reduce the space complexity with a negligible increase in time complexity.

Both the array-based and pointer-based tree representations can be easily extended to **k-ary** trees in which each node has exactly k children that we will refer to here as children 1, ..., k . For the array-based representation the ideas described for a binary tree are easily extended by observing that the j th child of node i is node $k \cdot i + j$, and the parent of node i is node $\lfloor (i - 1)/k \rfloor$. For the pointer-based representation, one could explicitly declare k child reference variables, but a more common design is to have an array of size k of child references. The advantage of this approach is a single array access can be used to access the j^{th} child of a node, versus conditional access

[†]If the Node class is not static then an additional reference (*this*) is included by Java for each node to maintain a reference to the tree that includes this node. Also, all objects contain a reference to their type.

to the appropriate variable. The abstract search tree (Chapter 31), B-tree (Chapter 36), B+-tree (Chapter 37), quad tree (Chapter 48), trie (Chapter 41), compact trie (Chapter 42), compressed trie (Chapter 43), and ternary search trie (Chapter 45) implement a pointer-based tree with an array of child references in each node.

The design becomes more complicated when the number of children of each node varies significantly over time. An array (or dynamic array) of child references can be used, with the advantage that any child of node i can be accessed in constant time. However, this can waste space since a number of the child references could be *null*. Another drawback of an array-based representation for child references concerns tree structure modifications that move a child reference from one node to another. In the worst case, the cost of this is proportional to the number of children in each node. If children references frequently must be moved between nodes, maintaining an array of child references is inefficient.

An alternate design is to let each node refer to a single child, and link all siblings together in a list. The pairing heap (Chapter 27) and Fibonacci heap (Chapter 28) use this approach, which has the advantage that a child reference can be moved from one node to a known location in another node in constant time. However, in general, reaching a particular child requires traversing the sibling list. Thus, in the worst-case, it takes time linear in the number of children to access a specific child of a node. While the sibling list always has the exact number of elements, the space usage is generally still higher than an array of child references unless a majority of the child references in the array are *null*. Based on the kinds of methods that must be supported for the ADT, there are additional design considerations for the list of children, such as:

- Should the sibling list have only a reference to the next sibling (**singly linked**) or also include a reference to the previous sibling (**doubly linked**)? The advantage of a singly linked list is the reduced space usage. However, if the ability to remove a node from the sibling list in constant time is needed, then a doubly linked list may be indicated.
- Should the list be **circular**, meaning that the last sibling in the chain references the one referenced by the parent? There is no additional space requirement. The advantage of a circular list is that the sibling referenced by the parent can change without any need for restructuring.

We close with one other tree representation which is appropriate for a conceptual design in which no methods are needed to access the children of a node. For example, suppose an application must maintain an organizational chart and the only operation needed is to follow the chain of command from a given employee to the CEO. In such cases a very good structural design is to use an **in-tree** in which the structure of the tree is captured exclusively through the *parent* references. Two data structures we present that use such a representation are the union-find data structure (Section 6.3) and the shortest path tree (Section 53.3). An in-tree makes a very efficient use of space, but traversals are limited to go from a node in the tree towards the root. Without adding additional structure, even iteration over the elements in the tree is not possible.

1.7 Further Reading

Gries [79] describes some of the early work on techniques to prove programs correct. According to Gries, Floyd [56] was the first to introduce the loop invariant. A good source on recent work in the area of program correctness is the book by Mitchell [114]. The use of an abstraction function as part of the process of understanding the correctness of data structures and algorithms is inspired by Liskov and Guttag [105]. For additional discussion of tree representations, see Knuth [96].

Chapter 2

Selecting an Abstract Data Type

Central to all software systems are data structures that provide methods for organizing and accessing information. Selecting appropriate data structures for an application is essential for both ease of implementation and efficiency of execution. This is because the efficiency of algorithms that operate on data depend heavily upon the organization that data structures provide. Therefore, the design of data structures and algorithms are closely intertwined. Object-oriented programming languages like Java capture this relationship through classes that provide both a data representation (e.g., a data structure) and methods that implement algorithms for operating on that representation.

Software developers sometimes make the mistake of choosing a data structure purely on the basis of what information must be stored. However, there are often many ways to represent the same information, and these alternatives may differ dramatically in how well they support the operations that the application may need to carry out. Therefore, as described in Chapter 1, it is best to begin by thinking about what kinds of questions the application will need to answer about the data, and then to identify (or perhaps define) an abstract data type (ADT) to support those operations. Only then does it make sense to begin selecting an appropriate implementation of that ADT, complete with the algorithms and supporting data structures.

Part II of this book is devoted to fundamental ADTs, and the data structures and algorithms that form their implementations. We use the term **application** to refer to the portion of a software system that will use the data structure in question. The application has certain requirements that it places on the data structure. These requirements form the basis of deciding which ADT is appropriate and which implementation of the ADT will be most efficient. We use the term **application programmer** to refer to the individual writing the software that will use the data structure, whereas the term **programmer** may also refer to the individual who has written or modified the data structure itself. Since our focus is on the ADT, we sometimes use the term **user** to refer to either the application or the application programmer making use of the data structure. This user is distinguished from the **end user**, the individual who will ultimately use the application.

The interface by which the programmer interacts with the ADT is defined by a set of methods (or procedures) that can be performed on the data. Methods fall in two categories: **mutating methods** (or **mutators**) that change the data structure and **non-mutating methods** (or **accessors**) that do not change the data structure but perform some computation to answer a question about the state of the data structure. We use the term **primary method** to refer to methods that characterize the ADT and must be efficiently supported by the ADT. The primary methods tend to be those that stand out as most natural in the way application designers would conceive of using the ADT. For example, using the event collection case study from Section 1.5, the method to find a list of events with a given date is a primary method, whereas a method to return the number of events in the collection is not a primary method. We use the term **operation** to refer to the computation performed by a method.

Selecting a data structure is a two-step process. The first step is to select a natural ADT, one that provides primary methods that best support the way in which the application will use the data. The second step is to select an efficient implementation of that ADT on the basis of the expected **access pattern**, the relative frequency with which each method will be performed. This chapter is concerned with selecting an appropriate ADT. The remainder of Part II is devoted to various

	USD	GBP	CAD	EUR	AUD
USD	1	0.557569	1.3048	0.826515	1.41563
GBP	1.79349	1	2.34015	1.48235	2.53893
CAD	0.7764	0.427321	1	0.633441	1.08494
EUR	1.20989	0.674602	1.57867	1	1.71276
AUD	0.706399	0.393866	0.921709	0.583849	1

Table 2.1 Exchange rates between 5 currencies on August 19, 2004 where each row in the table represents the exchange rate from that currency to each of the currencies represented by the columns.

implementations of these ADTs, with an emphasis on choosing the most efficient implementation for the expected access pattern.

For many applications, using a combination of the data structures presented in this book will suffice. However, custom software sometimes requires novel modifications of existing data structures. These modifications must be accomplished carefully in order to retain both the correctness and performance properties of the data structures. To inform such modifications, each implementation presented in Part II includes a discussion of correctness and an analysis of performance.

2.1 An Illustrative Example

A data structure captures relationships among data elements. The choice of an ADT certainly depends on the kinds of relationships represented. However, the primary methods provided by an ADT are often just as important as the nature of the data itself in selecting the ADT. The following example illustrates some of the issues involved in ADT and data structure selection for two applications that use similar information in different ways.

Our example considers applications that must access monetary exchange rate information between the US dollar (USD), Great Britain pound (GBP), Canadian dollar (CAD), Euro (EUR), and Australian dollar (AUD). One way to represent the information is shown in Table 2.1, where each row of the table represents the exchange rate from that currency to each of the currencies represented by the columns. For example, each Euro unit will be exchanged for 1.2099 US dollars.

Application 1: We first consider an application in which the primary method required is *lookup*, which returns the exchange rate between two currencies. More specifically, *lookup(A,B)* should return the exchange rate from *source* currency *A* to *destination* currency *B*.

The table holding the exchange rates could be stored using a data structure with an internal representation of a **two-dimensional array**. A nice feature of a two-dimensional array is that in constant time the entry in row *i* and column *j* can be accessed.

In looking at Table 2.1, it might appear that one step is sufficient to look up the exchange rate. However, imagine that Table 2.1 had thousands of currencies. To perform *lookup(A,B)*, the implementation must first determine which row corresponds to currency *A* and which column corresponds to *B*. If the *n* currencies are arranged alphabetically, then the binary search algorithm, discussed in Section 30.3.3, could be used to find the desired row and column of the array in logarithmic time. If the currencies are not arranged alphabetically, then a linear time search through all rows and columns may be required.

To improve the efficiency of *lookup(A,B)*, one could augment the representation of Table 2.1 by adding a **mapping** to efficiently map the three letter acronym used for each currency to a

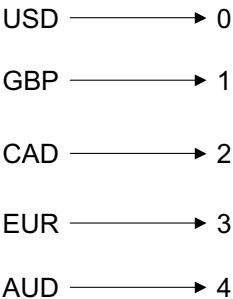


Figure 2.2
Pictorial view of a mapping from the three letter acronym for the currency to the row/column number of Table 2.1.

row/column number where we assume the top row is row 0 and the leftmost column is column 0. In particular, since the three letter acronym serves as a distinct identifier (or **key**) for the currency, the **Mapping ADT** could be used. In our example, we would create the mappings: USD $\rightarrow$ 0, GBP $\rightarrow$ 1, CAD $\rightarrow$ 2, EUR $\rightarrow$ 3, and AUD $\rightarrow$ 4, as shown in Figure 2.2. Now given two currencies, a good implementation of a Mapping ADT can be used to determine in constant time the row and column numbers of the array entry that holds the desired exchange rate. Then in constant time the exchange rate can be looked up in the array. Thus, by combining a mapping implementation and a two-dimensional array, $lookup(A,B)$ can be implemented in constant time.

Another way to use a mapping to implement $lookup(A,B)$ is to store the exchange information directly in a mapping of mappings. The primary mapping would map the source currency acronyms to secondary mappings that, in turn, map the destination currency acronyms to the exchange rate from the source to the destination currency. Figures 2.3 and 2.4 show this pictorially in two different ways. If we had an object *exchangeRate* that held this mapping of mappings, where the method *get* looks up the mapping, then *exchangeRate.get(A).get(B)* would implement $lookup(A,B)$. The Mapping ADT is discussed in depth in Section 49.7.

Application 2: We now consider a different primary method an application might require. **Arbitrage** is the act of converting one unit of currency A to another currency, and that currency to yet another currency, and so on, until returning to currency A in such a way that the result is more than one unit of currency A. Instead of simply looking up the exchange rate between two currencies, suppose our application needs to determine if an arbitrage scheme exists.

For this problem, using a **graph** representation (Chapter 52) leads to the most efficient solution. The exchange rate data can be represented using the graph shown in Figure 2.5, where each currency is represented as a **vertex**, and each possible exchange is represented as an **edge**. More specifically, an edge from vertex *u* to vertex *v* has a **weight** which is the exchange rate when converting from currency *u* to currency *v*. In this example, the graph is a **complete graph** which means that there is an edge (in each direction) between each pair of vertices. In general, a graph need not include all possible edges.

More generally, one can view a graph as a set of objects (the vertices) and a set of associations between the vertices (the edges), where a weight may be associated with each edge. A **path** in the graph is a sequence of edges that leads from one vertex to another. A **cycle** is a path that starts and ends at the same vertex. The kinds of methods that can be most efficiently performed using a graph representation are those that involve paths in a graph. For example, solving the arbitrage problem is equivalent to determining if there is a cycle in the graph such that the product of the edge weights along the cycle is greater than 1. We return to this example in Section 57.7 and show how it can be

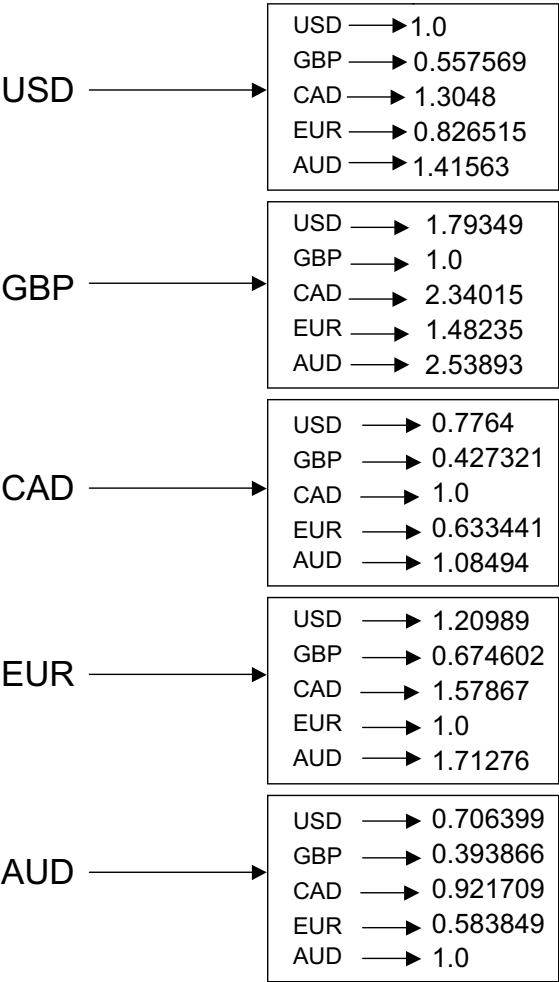


Figure 2.3
Pictorial view of a data structure to look up an exchange rate that uses a mapping of mappings for the exchange rates shown in Table 2.1. The first mapping is from the source currency to a secondary mapping shown enclosed in a rectangle. Each of the secondary mappings is from the destination currency to the exchange rate.

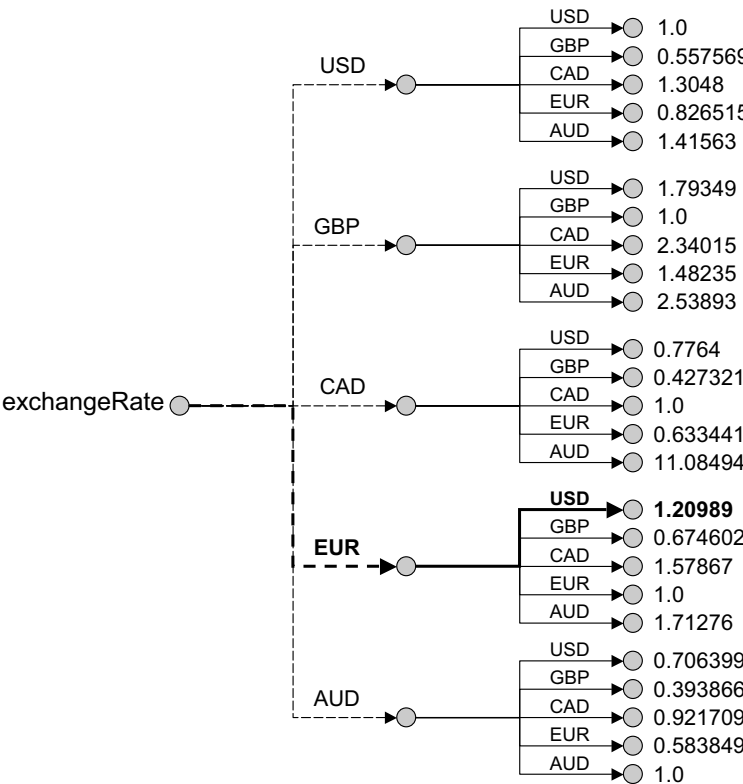


Figure 2.4
An alternate pictorial representation for the relationship of Figure 2.3. The primary mapping from the source currency to the secondary mapping is shown using a dashed line style. The secondary mapping is shown as a solid line. The thick lines show the mappings that would be used to look up the exchange rate between the Euro and the US dollar.

formulated as a shortest path problem. Because the arbitrage problem is most naturally expressed as a question about paths, a graph is a more natural choice than an array or mapping.

In summary, each of the three views for the exchange rate data provide a different way of looking at the data relationships. Although these views look different, they represent the same information. For each entry in Table 2.1, there is one edge in the mapping of mapping view (Figures 2.3 and 2.4), and one edge in the graphical view (Figure 2.5). Therefore, one might conclude that the choice of ADT is arbitrary. However, this is not the case. In general, it is more natural to pose certain questions and perform certain methods under one representation than another. As the above examples illustrate, the set of primary methods is a key factor in selecting an ADT and must be considered in conjunction with the most “natural” representation of the data.

2.2 Broad ADT groups

In the remainder of this chapter, we discuss three major groups of ADTs: partitions over a set of elements, collections of elements (where the elements could be any object including *null*), and graphs.

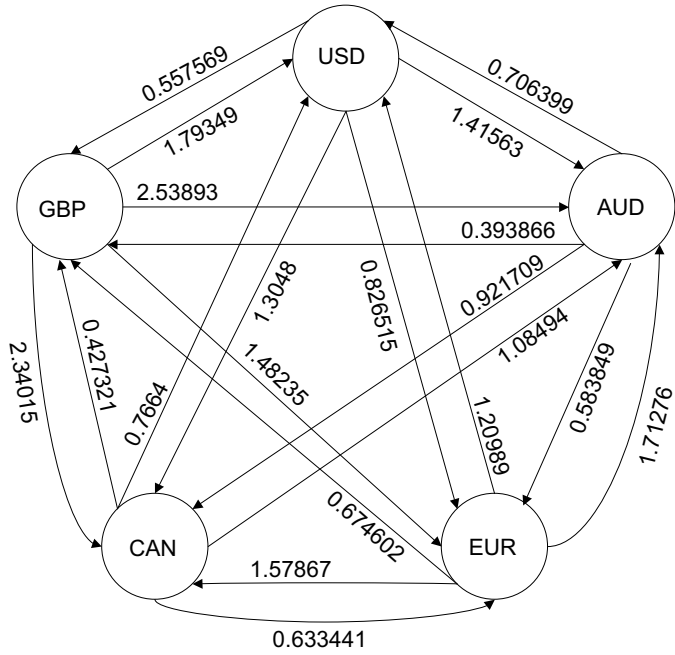


Figure 2.5
Graph representation of the exchange rates shown in [Table 2.1](#).

A **partition** is a division of a group of objects into any number of disjoint sets. For example all animals are partitioned into a set of species where each animal belongs to exactly one species. As another example, suppose a city is installing bike paths between some of the city parks. One can define an **equivalence relation** over the parks, where each set in the partition includes the parks mutually reachable from each by bike paths. We view a partition abstractly as a collection of disjoint sets (or components) whose union is the total collection. However, the actual data type we use to represent the partition is not the partition as a whole but rather the individual components of the collection. So the data type that we implement for the Partition ADT is that of a *PartitionElement*, introduced in Section 2.3.

The most commonly used class of ADTs are those that maintain a **collection** of elements. Many of the ADTs provided as part of the *java.util* library maintain collections (e.g., List, Map, NavigableMap, Queue, Set, SortedMap, SortedSet). As a result, this broad group of ADTs is the one that is most familiar to many application developers. However, in Section 2.4 we introduce a much richer set of ADTs for maintaining a collection than those provided within the Java libraries.

Finally, a **graph** is commonly used to capture **binary relations**, relations among pairs of objects. For example, a set of parks connected by bike paths could be modeled as a graph in which there is a vertex corresponding to each park, and an edge between any two parks that are directly connected by a bike path. As discussed in the earlier examples of this chapter, a graph is particularly well suited for applications where the fundamental methods relate to paths within the graph. Graphs are discussed further in Section 2.7.

The next three sections discuss these groups of ADTs in more depth to provide further guidance in selecting among them for a given application. We also discuss ADTs contained within each group and how to select among those. Since the choice of the best ADT is sometimes subtle, and since many applications require a combination of more than one type of ADT, familiarity with all three groups will lead to better decisions.

2.3 Partition of a Set

A partition is a very specialized ADT in which a set of elements is divided into disjoint subsets such that each element is in exactly one subset. More formally, given a set S , the subsets $S_1, S_2, \dots, S_k$ define a **partition** of S if and only if $\bigcup_{i=1}^k S_i = S$ and for all $1 \leq i < j \leq k$ $S_i \cap S_j = \emptyset$. We refer to each subset of the partition as a **component**. Rather than introduce a node for each component, we let each component have a **representative element** by which that component is identified.

The two primary methods of the *PartitionElement* interface are:

union(S_i): combines S_i and the partition element on which this method is called into one component, and

findRepresentative(): identifies a representative (or canonical) element for the group to which this partition element belongs.

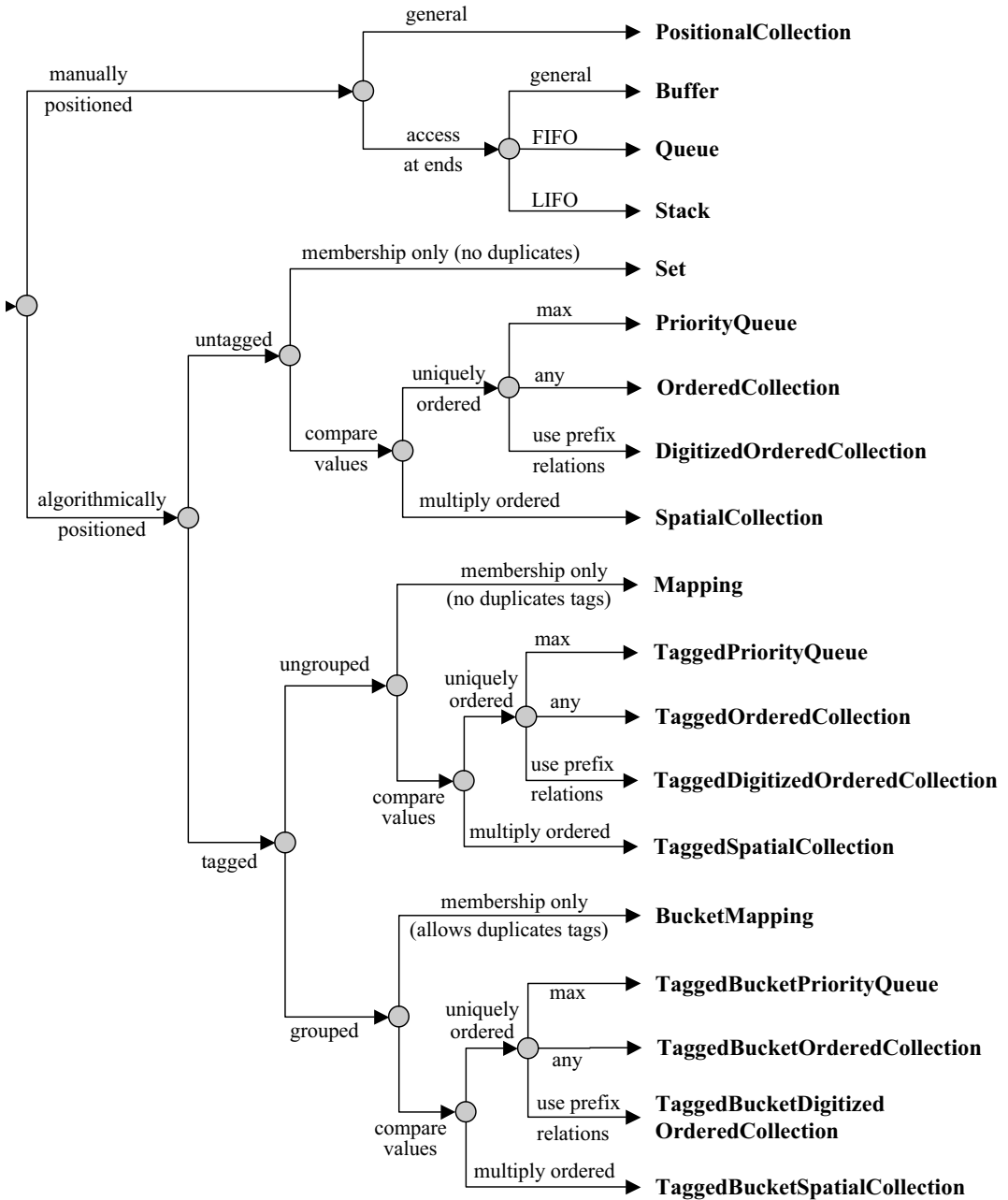
We use a partition for two very different applications. In Section 6.8 we use the Partition ADT to support the ability to merge two set data structures such that data structure objects from both of the original sets can be reused and locators remain valid. The overhead introduced is nearly constant, so a constant time merge is possible when the underlying data structure supports it. Initially, each data structure is a singleton component. Whenever two data structures are merged, the components to which they belong are merged with the *union* method. The representative element for each component maintains the instance variables shared by the entire component. The *findRepresentative* method can be used to determine if two data structures are in the same component by determining if they have the same representative. See Section 6.8 for more details.

A second application for the Partition ADT is to maintain a set of connected components in a graph, discussed in the example of placing bike paths between parks. Kruskal's minimum spanning tree algorithm (Section 57.4) uses the Partition ADT for this purpose.

2.4 A Collection of Elements

Many applications must maintain a collection of elements. However, applications use collections in different ways, so we discuss different ADTs that support a variety of options for creating and using collections of elements. This section provides guidance for selecting among these options. [Figure 2.6](#) summarizes the selection process as a decision tree.

The first consideration in selecting a collection ADT is how often, and by what means, an application will need to locate an element within the collection. In some cases, an element can be located by its position in a list. In other cases, an element with a specified value is desired. Although one could certainly locate an element with a specified value in a data structure using a brute force linear time search that considers every element in the data structure, most applications require more efficient access to elements. The first consideration in selecting an ADT is to ensure that the ADT supports methods for accessing elements using information the application can provide to identify them. Then among the competing ADTs that satisfy this requirement, the right choice is the one whose other primary methods best support important (frequent) activities of the application in a natural way.

**Figure 2.6**

A taxonomy of ADTs that maintain a collection of elements.

2.5 Markers and Trackers

Sometimes an application will retain (directly or indirectly) a reference to the part of the data structure that holds the desired element. When the application presents this reference to a method of the data structure, the desired element can be found in constant time, simply by following the reference. For example, suppose a linked list data structure consists of a number of list items, each with a reference to an element in the collection and a reference to the next list item. If the application retains a reference to a list item for a particular element, it can present that reference to a method of the data structure to access the corresponding list item in constant time.

In most cases, encapsulation demands that the application not be given a direct reference to objects within the data structure. Otherwise, the application might inadvertently violate important properties of the ADT implementation by incorrectly manipulating the object. For example, the application might truncate a linked list by incorrectly setting the next reference of a list item to *null*. Furthermore, exposing the internal representation to the application would break the abstraction barrier and therefore prevent making future changes to the internal representation of the ADT implementation.

Nonetheless, it is sometimes desirable for an application to retain references that can be used by a data structure to provide constant time access. To avoid exposing the internal representation of the data structure, we encapsulate such references inside of objects. Such objects can be retained and used by the application, but the encapsulated reference can be used only by the internal implementation of the data structure. An **iterator** is one example of such an encapsulated reference. The application retains a reference to the iterator, which in turn contains a (hidden) reference to an object and/or position within the data structure. By advancing the iterator one can iterate through a collection to visit each element exactly once. The order in which the elements are visited is called the **iteration order**. We define **FORE** to be logically just before the first element in the iteration order, and **AFT** to be logically just after the last element in the iteration.

The collections in this book usually provide one of two different types of encapsulated references. Both types extend the Java *Iterator* interface, so either can be used to traverse a collection. (See Section 5.8 for a more complete discussion.) However, the two types are distinguished in the way they behave when the collection is modified.

- A **Marker** holds a position within the data structure. For example, a particular marker might refer to the third position in a data structure representing some sequence. If a new element is added at the beginning of the sequence, the marker would still refer to the third element, the one that had previously been second in the sequence. (If the marked position no longer exists, then the marker is placed at the special value **AFT**.)
- A **Tracker** encapsulates a reference to the part of a collection that holds a particular element. Even in the face of modifications to the data structure, such as adding and removing elements, the tracker will continue to refer to the part of the collection that holds that element. If a tracked element is removed, then the tracker is logically between the two neighboring elements in the iteration order, with **FORE** considered as the start of the iteration order, and **AFT** as the end.

Markers and trackers provide constant time access to elements in a collection. They both provide support for navigating within the collection, as well as inserting and removing elements at the current position.

2.6 Positioning and Finding Elements

Locators are useful when iterating over a collection, and whenever constant time access is needed for known elements in the collection. However, applications often need to access elements in data structures on the basis of other information. The application may not even have knowledge about which element is needed, let alone a marker or tracker to that element. For example, an application may need to access the element of a collection with the highest scheduling priority. Or perhaps it needs to find some data associated with a particular person (based on their name). Markers and trackers provide no help in such cases. Instead, the data structure must provide methods that support access to elements on the basis of semantic information about the desired element or the about relationship of the desired element to the other elements in the collection. We define **semantic information** to be any information contained within the data that has meaning at the application level. For example, the alphabetical order used to organize a phone book.

We summarize these ideas by example. Consider a data structure that maintains a printer queue. The most common methods used would be to insert a new job at the back of the queue or get the job at the front of the queue. Both of these methods use semantic information. In this case, the semantic information concerns the relative ordering of the elements within the queue. Another required method might allow a user to cancel a particular job in the queue. If this is to be performed without searching through the entire queue, the user could retain a tracker to the element and then provide it to the cancel method. Alternatively, the user could provide the name of the print job, which could be used as a key in a mapping from job ids to trackers. Finally, one may want to iterate through the queue to list all of the jobs. If the queue is not being modified concurrently, this operation could be carried out using either a tracker or a marker. However, if concurrent modification is permitted, a tracker should be used so that the current location is unaffected by position shifts that occur as elements are added or removed from the queue.

In selecting among collection ADTs, one factor to consider is whether the application can take advantage of (encapsulated) references to the data elements, as described in Section 2.5. However, such references can only improve performance for certain common operations, and applications generally do not retain references for every element in a collection. A central role of a collection often is to provide access to the data elements on the basis of semantic information about the elements or their relationship to other elements in the collection. *Therefore, in selecting an ADT, a key factor is the form of the semantic information that an application will use to identify the desired element.* This semantic information falls into two general categories.

Manually Positioned: The user views all of the elements as being stored in a line of n elements, and identifies an element by its position in this line, where positions are identified by integers from 0 to $n - 1$. By providing a position within the line, the user can call methods to place a new element at beginning of the line, place a new element at the end of the line, replace an element at the specified position, or insert a new element after the element at the specified position. Similarly, the user can request removal of the first or last element in the line, an element at a given position, or a particular element whose position is unknown. The data associated with a given position can be *null* as a way of indicating that nothing is in that position. Such *null* entries are treated as elements of the collection.

Algorithmically Positioned: The user views the elements as logically organized based on some property (typically equivalence or an ordering among the elements). The user does not directly control the placement of the elements. We further subdivide this category into the following three groups.

Untagged: The property used to organize the elements is directly defined over the elements themselves. For example, the user may want to efficiently determine if there is an element in the collection equivalent to a provided element. Another possibility is that the elements in the collection are comparable and the logical organization desired by the user is based on the relative order among the element values. For example, the application program might want to find the element that has a maximum value, or all elements that fall into a given range of values.

Tagged: The property used to organize the elements is an external **tag** that is associated with each element. A **tagged element** is a tag-element pair. Although the tag may be a field in the representation for the element, typically the tag is not part of the element. However, sometimes it is more natural for the application to specify the tag separately even if the tag is part of the element. For example, it may be more natural to tag a product with its product id, rather than define a comparator that extracts the id from a product to perform comparisons. Also, when a large collection of large elements is kept on secondary storage, one can minimize disk accesses by creating a separate index as a tagged collection that duplicates the (relatively small) tag information from each element and associates with it the disk location of the element.

The element (sometimes called **data object**) that is associated with the tag can be *null*, an object, an entire data structure, or an offset into an external file. Each insertion into a tagged collection ADT creates an association between the tag and data object. Once the tag is associated with each element, the user can access the tags in the collection in ways similar to elements of an untagged collection, with the additional benefit of finding the element(s) associated with a given tag.

When the tag uniquely identifies the element it is called a **key**. For example, a student id might be used as a key, where the associated data is the entire student record (which would include contact information, a transcript, and so on). *Although keys are unique, tags need not be.* For example, an application to control a print queue might tag each job with a priority, where many jobs might have the same priority.

Tagged collections are further divided based on whether all the elements associated with the same tag are stored as separate tagged elements or are grouped together in a tagged **bucket**.

Ungrouped: When tagged elements that have the same tag are kept as individual entities, we define such an ADT as a **tagged collection**. Each insertion into a tagged collection introduces a new tagged element. For example, if tagged elements $A \rightarrow 10$, $A \rightarrow 5$, $B \rightarrow 12$, $A \rightarrow 7$, and $B \rightarrow 6$ are inserted into some tagged collection, then the tagged collection will hold 5 tagged elements. Although a tagged collection is organized to efficiently find a tagged element with a desired tag, it would be necessary to iterate through the collection to find all the elements with a given tag. In a tagged collection the cost associated with locating a desired tagged element generally depends on the number of tagged elements in the collection

Grouped: When all tagged elements associated with the same (or equivalent) tag are grouped together, we define such an ADT as a **tagged bucket collection**. Like a tagged collection, a tagged bucket collection also uses the tag to organize the elements. The important difference is that in a tagged bucket collection, a set of elements that share a tag are stored together in a **bucket** (or set). Each insertion into a tagged bucket collection ADT also creates an association between the tag and the data object. If any data object is in the collection that has tag t , then a bucket associated with tag t holds all of the data objects associated with t . For example, if tagged elements $A \rightarrow 10$, $A \rightarrow 5$, $B \rightarrow 12$, $A \rightarrow 7$, and $B \rightarrow 6$ are inserted into a tagged bucket collection, then it would hold the two buckets $A \rightarrow \{10, 5, 7\}$ and

$B \rightarrow \{12, 6\}$. A bucket tagged collection is organized to support efficient access of the collection of data objects with an associated tag. For a tagged bucket collection, the cost of locating a desired tag generally depends on the number of unique tags. As an example, consider the problem of indexing a document to efficiently locate all occurrences of a given word in the document. A natural approach is to create tagged elements in which the tag is the word in the document, and the associated data is the offset into the document for an occurrence of the word. Suppose that the document had 50,000 words but only 5,000 distinct words. (So on average each word occurs 10 times). In a tagged collection, there would be 50,000 tagged elements, and the query of finding all locations for a given word is not very naturally supported. In contrast, in a tagged bucket collection, there would be 5,000 tags (one for each distinct word), each associated with a bucket with all offsets in the document for that word. In general, when the application tends to associate many elements with the same tag, a tagged bucket collection is a better choice than a tagged collection.

The remainder of this section provides guidance for selecting among manually positioned ADTs, untagged algorithmically positioned ADTs, tagged algorithmically positioned ADTs, and tagged bucket algorithmically positioned ADTs.

2.6.1 Manually Positioned Collections

The primary distinction among the ADTs in this group is which positions the user can access without a locator.

General: For some applications, the user needs the ability to access, add, and remove elements at any position in the collection. For such applications, the **PositionalCollection ADT**, the most general manually positioned collection, is the right choice.

Access only at the ends: For many applications, elements need only be added or removed from the front or back end of the collection. Data structures designed for such settings can gain efficiency by limiting access to the ends. The most general ADT for this category is the **Buffer ADT** in which the user can only insert or remove an element from either end of the collection. The **Queue ADT** maintains a first-in, first-out (**FIFO**) line. Elements can only be inserted at the back of the line and removed from the front of the line. The **Stack ADT** maintains a last-in, first-out (**LIFO**) line. A stack is logically viewed as a vertical line in which elements are inserted and removed at the “top” of the stack. The Buffer, Queue, and Stack ADTs can be **unbounded**, meaning that there is no limit to the number of elements held in the collection, or **bounded**, in which case there is a limited size. An *AtCapacityException* is thrown if an insertion is attempted on a bounded collection that is already at its capacity.

2.6.2 Algorithmically Positioned Collections

There are two orthogonal decisions for applications that need an algorithmically positioned collection. The first is whether the application can directly organize the elements using their values (**untagged**) or whether it is necessary, or perhaps just more convenient, to add a tag to organize the elements (**tagged**). The choice between the corresponding tagged and untagged collection is determined by whether the equivalence tester (and if appropriate the comparator) is a function of field(s) in the object, or a function of externally imposed information.

When the elements are organized based on a tag associated with each element, the next decision is whether or not the elements with the same associated tag should be stored as individual tagged elements (**ungrouped**) or combined into one bucket associated with the shared tag (**grouped**). The advantage of grouping elements with a shared tag is that the search cost depends only on the number

of unique tags, versus the number of elements, and grouping the elements supports efficient access to all elements with a given tag. For example, consider a very simple event recording application that tags each event with the date when it occurred, where the application program needs a way to find the set of events that occurred on a given date. Generally, when it common for more than just a small number of elements to share a tag, a tagged bucket collection that groups the elements is a better choice.

Regardless of the choice of using an **untagged collection**, a **tagged collection** (ungrouped), or a **tagged bucket collection** (grouped), the rest of the selection process is the same with the only distinction being whether the organization is based on the element or the tag, and whether the elements with the same tag are ungrouped or grouped. The important distinguishing criterion among the ADTs for algorithmically positioned collections is how the elements/tags define the organization used by the application, and what properties they require to support the computations required by the application.

Is the element/tag only used to test for membership? For these applications, the elements/tags define a set in which there are no duplicates. The elements/tags must have a notion of equivalence, but need not be comparable. The elements/tags may happen to be comparable, but no operation depends upon their relative order. When the application must efficiently determine if a particular element is held in the collection, the **Set ADT** is the best choice. When the application must locate the data object (if any) identified by a given (unique) tag, the **Mapping ADT**, a tagged collection, is the best choice. Finally, when the application must locate the set of data objects (if any) associated with a given tag, the **BucketMapping ADT**, a tagged bucket collection, is the best choice.

Do the primary methods compare the elements/tags? The next group of ADTs for algorithmically position collections are those in which the methods depend upon some ordering among the elements/tags as specified by either a default comparator or a user-provided comparator. We further divide this group of ADTs based on the following questions.

Are the elements/tags uniquely ordered? We say that a set of elements/tags are **uniquely ordered** when there is a unique answer to whether element/tag a is less than element/tag b for any two non-equivalent elements/tags a and b . Many applications have elements/tags that are uniquely ordered. When this ordering is important to the organization of the data required by the application, one of the following ADTs is most appropriate:

Priority Queue - This collection is the appropriate choice when the comparator defines an ordering of the elements/tags that corresponds to a priority, and the primary methods use the ordering to efficiently locate the element/tag with the highest priority. (A **Tagged Priority Queue** is the corresponding tagged collection, and the **Tagged Bucket Priority Queue** is the corresponding tagged bucket collection.)

Ordered Collection - This collection is the usually the best ADT when the iteration order and methods depend upon the total order of the elements/tags defined by the comparator. However, if it is possible to view elements/tags as a sequence of digits, then the **DigitizedOrderedCollection ADT** should also be carefully considered. (A **Tagged Ordered Collection** is the corresponding tagged collection, and the **Tagged Bucket Ordered Collection** is the corresponding tagged bucket collection.)

Digitized Ordered Collection - This ADT is an ideal choice when the elements/tags can be viewed as sequences of digits, especially when the application requires finding the set of elements/tags that share a provided prefix or finding the elements/tags

in the collection with the longest common prefix of a specified sequence. This ADT also has the advantage that the search time is upper bounded by the length of the prefix of the target that distinguishes it from other items in the collection. Thus, for a very large collection containing many elements/tags with a small number of digits, a digitized ordered collection is most likely the best choice. (A **Tagged Digitized Ordered Collection** is the corresponding tagged collection, and the **Tagged Bucket Digitized Ordered Collection** is the corresponding tagged bucket collection.)

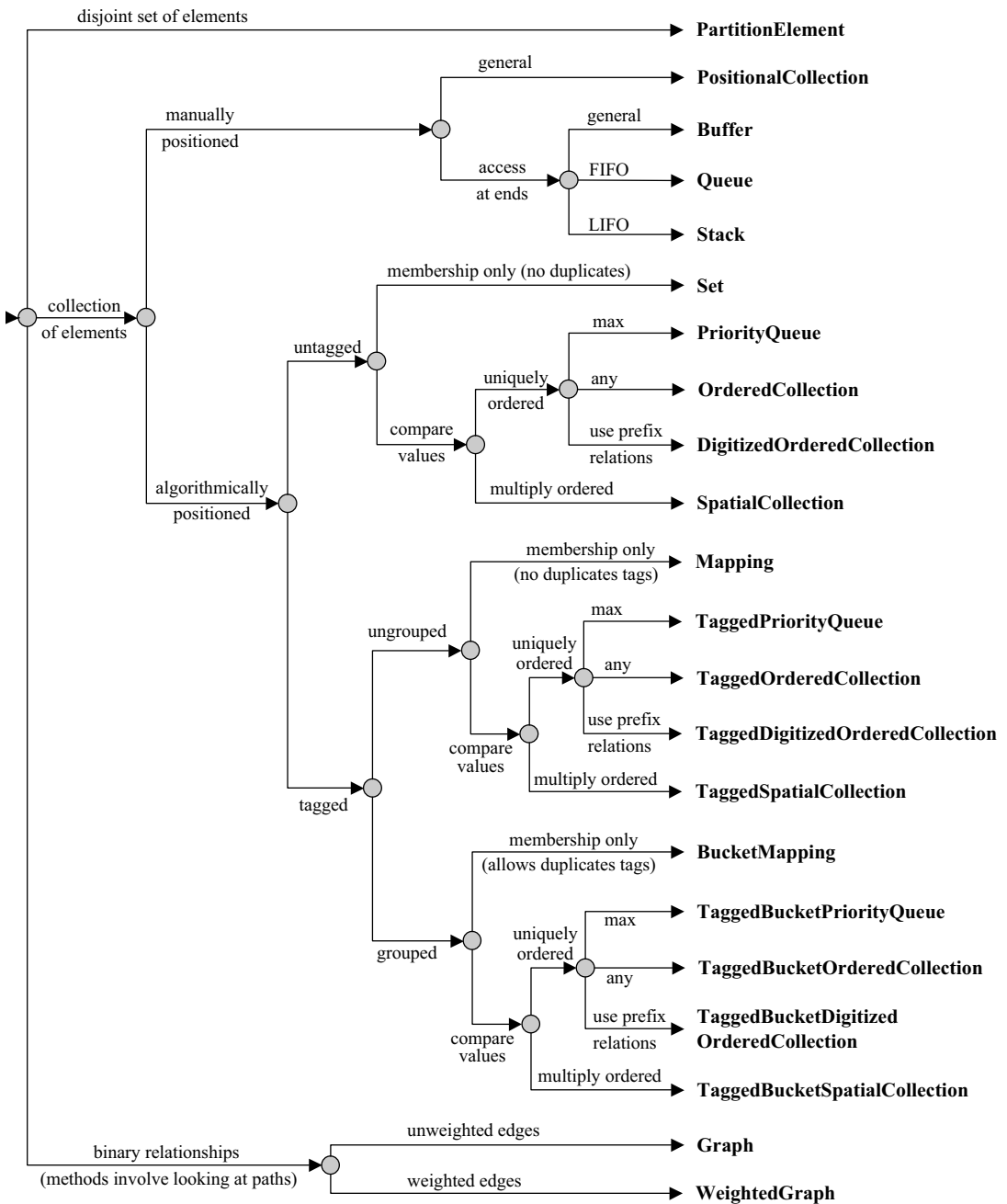
Are the elements/tags multiply ordered? We say that a group of objects is **multiply ordered** when multiple criteria define different total orderings of the objects. Each of these criteria can be implemented as a different comparator. For two objects a and b , one ordering may consider a less than b , while another ordering may consider b less than a . For example, an application that maintains the locations for a set of mobile devices might need to efficiently find all devices that were within a rectangle defined by a longitude and latitude range during a specified time interval. Each device object could have instance variables for the longitude, latitude, and time that are used to define three distinct orderings. Geometrically, the query given above corresponds to finding all devices (viewed as a 3-dimensional point based on the longitude, latitude, and time) that are in the axis-aligned box defined by the specified ranges for longitude, latitude, and time. The **SpatialCollection ADT**, **TaggedSpatialCollection ADT**, and **Tagged-BucketSpatialCollection ADT** is designed for such applications.

Figure 2.7 provides a complete decision tree to guide the process of selecting among all ADTs discussed in this book. A simplified version of this decision tree also appears inside the front cover, and corresponds to the tabs in the right margin throughout the book. The portion of that decision tree under the “collection of elements” branch is the decision tree for collections shown in Figure 2.6.

2.7 Graphs

A graph represents general relations between pairs of elements from among a set of elements. The elements are called **vertices**, and the relationships between them are called **edges**. Many real-world algorithmic problems can be formulated as graph problems. One example we have already discussed in Section 2.1 is arbitrage. Another example is commercial flight information between a set of airports. Here, a vertex represents an airport, and an edge represents a flight. (See the case study presented in Section 56.1.) Unlike the arbitrage example, the flight graph is not a complete graph: there are some pairs of airports with no direct flights between them. If the only task required is to look up flight information then a mapping could be used. However, typical methods for this application are to find a sequence of flights from airport A to airport B using the fewest flights, or perhaps using the least time, or least cost. All such methods involve paths in the graph, so a Graph ADT is best.

A Graph ADT is not necessarily appropriate for capturing all types of relationships among data elements. In some cases, relationships are intrinsic, computable from the elements themselves. Such relationships need not be represented explicitly, and are therefore *not* generally appropriate for representation as edges of a graph. Examples of these kinds of relationships include: “Is element A smaller than element B?” where smallness can be determined or computed on the basis of values stored inside the elements, and “Is string A a prefix of string B?”

**Figure 2.7**

A decision tree for selecting an ADT.

More generally, a graph is not appropriate when the relationship among elements involve some intrinsic order and the primary methods relate to this order. For such applications, the **OrderedCollection ADT**, **TaggedOrderedCollection ADT**, **DigitizedOrderedCollection ADT**, or **Tagged-DigitizedOrderedCollection ADT** is the best choice. See [Section 2.4](#) for further discussion.

Recognizing when to choose a Graph ADT over some other ADT involves subtle distinctions since many ADTs capture relationships among elements. To help recognize when to use a graph ADT, consider the following questions in the context of your application. A “yes” answer to one or more of the questions is a strong indication to use a Graph ADT.

Are the elements connected by a network of (existing or potential) links, such as roads, flights, pipelines, or wires?

One of the most common situations in which to apply graph algorithms is when the vertices of the graph represent physical locations such as intersections, airports, or switches, and when the edges represent connections between them such as roads, flights, or wires. In such cases the fundamental methods almost always relate to paths in the network, such as finding the shortest way to get from one vertex to another, or finding the least cost set of edges to ensure all vertices are connected. The arbitrage problem discussed earlier can actually be formulated in such a manner where the vertices are the currencies, and the edges represent an exchange of money from one currency to the another.

Is there a precedence relation among the elements? (For example, Task A must precede Task B, or Course A must be taken before Course B.)

A graph created for such data is often called a **precedence graph**. The edges in the precedence relation are not intrinsic, meaning that they could not be computed by a comparator over the elements. The method most commonly required is to find an order in which to perform the tasks that does not violate any of the precedence relations. A graph is the right choice here since transitive relations (e.g., if Task A must precede Task B, and Task B must precede Task C, then it follows that Task A must precede Task C) correspond to paths in the precedence graph.

Is the data hierarchical in nature, and do primary methods involve the structure of the hierarchy?

An example of data that is hierarchical in nature is an organizational chart where the CEO is at the top of the hierarchy. All employees who directly report to the CEO form the second level of the hierarchy, and so on. Another example is a taxonomy of species. A graph in which there are no cycles is called a **tree**. Furthermore, if there is a distinguished “top” or root of the tree, it is called a **rooted tree**. (See [Section 1.6](#) for further discussion on rooted trees and their representations.) When data is hierarchical in nature it is often best represented by a rooted tree. As with the other examples, one could represent the tree for an organization chart by a mapping which has relations of the form “Person A directly reports to Person B.” If the user only needs to answer questions of the form, “To whom does Person A report?” then a mapping can be used. If the only fundamental question is to determine if two people are in the same department of a company, then the Partition ADT might be best.

However, when data is hierarchical, methods that relate to the graph structure (and more importantly paths within the structure) are often needed. For example, one might want to find the person lowest in the hierarchy who is directly or indirectly responsible for a set of company employees. When such methods are required, a graph representation (with the special case of a rooted tree) is the best choice.

Are all elements being related of the same type?

To understand this question, it helps to view the relationships pictorially. The relationships shown in [Figure 2.2](#) are between objects of different types since it is a mapping from an acronym to a number. When the answer is “no” to this question, it is likely that the Mapping ADT is appropriate.

See Section 49.7 for much more in-depth discussion of the Mapping interface and the ADTs that implement a mapping. The relationship shown in [Figure 2.3](#) also creates a mapping between objects of different types. The first maps from a currency acronym to another mapping, and the second set of mappings, map from a currency acronym to a double (the exchange rate).

In contrast to this example, notice that in the representation shown in [Figure 2.5](#), the relationships always map one currency to another currency. When the related elements are the same type, it is sensible to consider a graph.

Are some elements related to more than one other element? In other words, would some vertices have two or more outgoing edges?

This question is also best understood by viewing the relationships pictorially. In the relationships shown in both [Figures 2.2](#) and 2.3, not only are related elements of different types, but also there is only one edge from each vertex. Typically, graphs are useful only when some elements are related to more than one other element, which corresponds to some vertices having at least two outgoing edges. When there is a single edge from each vertex, a Mapping ADT is likely to be the appropriate choice.

While the relationships shown in [Figure 2.4](#) are drawn with each vertex having five outgoing edges, really the graphical view is a “shorthand” for the six mappings shown more explicitly in [Figure 2.3](#). There is one mapping from the source exchange acronym to the secondary mapping, and the five secondary mappings from the destination exchange acronym to the exchange rate. One way to recognize this structure is to observe that the edges are going from an element of one type to an element of another type. Because the elements are not of the same type, this is a strong indication that a graph is not the right choice. In contrast to this example, notice that in the representation shown in [Figure 2.5](#), each vertex has five outgoing edges, each from a currency acronym to another currency acronym.

Apart from relationships that are directly computable from the elements themselves, are there explicit relationships, either asymmetric or symmetric, among the elements?

Some explicit relationships are directional (asymmetric). A **directed graph** is used when the relationship is asymmetric. The direction of each edge indicates how the relationship is applied. Other explicit relationships are symmetric (i.e., the two elements can be interchanged arbitrarily). An **undirected graph** is used when the relationship is symmetric. To illustrate the distinction, We describe four example applications that are best modeled as a directed graph. Then we describe two example applications best modeled as an undirected graph.

Currency Exchange Problem: Given the exchange rate between all pairs of currencies, compute the way to convert from currency A to currency B that provides the best exchange rate. (Without the fee typically charged to exchange currency, the direct exchange from A to B might not yield the best exchange rate.) Since the exchange rate from currency X to currency Y is different from the exchange rate from currency Y to currency X, the provided data is best modeled as a directed graph like that shown in [Figure 2.5](#). Observe that finding the best way to convert from currency A to currency B involves composing a sequence of exchanges that corresponds to traversing a path from A to B in the resulting directed graph. This problem can be modeled as a **shortest path** problem. Algorithms to compute shortest paths are discussed in Sections 57.2 and 57.7.

Least Cost Flight Problem: Given all flight information (e.g., flight number, airline, departure and arrival times, fare, etc.) for all commercial flights, find the least expensive itinerary to fly from airport A to airport B. Since the existence of a flight from airport X to airport Y at a certain time and cost does not imply a flight from airport Y to airport X at the same time and same cost, the flight information is best modeled as a directed graph with a directed

edge from X to Y for each flight from airport X to airport Y. The resulting graph will have multiple directed edges between vertices since there are generally many flights between two airports. A directed **multigraph** is the term used for a graph with multiple edges from one vertex to another. Observe that finding the least cost flight from airport A to airport B involves composing a sequence of flights. This corresponds to traversing a path from A to B in the resulting directed graph. Again, this problem can be modeled as a shortest path problem.

Predator Problem: Given information about whether each species is a predator for each other species, determine if species A is above species B in the food chain. Since species X being a predator for species Y is different than saying that species Y is a predator for species X, the information about the predator relationships between all species is best modeled as a directed graph where there is an edge from X to Y if and only if species X is a predator for species Y. In this example, the edge represents boolean information: it is either there or it is not. When the edges hold boolean information, the resulting graph is an **unweighted graph**. Observe that species A is above species B in the food chain if and only if there is a directed path from A to B in the directed graph representation. **Breadth-first-search** which is discussed in Section 53.4 can be used to solve this problem.

Maximum Flow Problem: Given information about topology and operating characteristics of a pipeline (including the capacity of each pipe and switch), determine how many liters per second can be shipped in an oil pipeline from location A to location B. In this case, the oil can only flow in one direction along any given pipe, so the topology and operating characteristics of the pipeline are best modeled as a directed graph where there is a vertex for each switch and location and a directed edge from X to Y when there is a pipe from location/switch X to location/switch Y. (There could be two pipes in opposite directions, possibly with a different capacity, that would be modeled as two distinct directed edge.) Determining the maximum rate at which oil can be shipped from location A to location B in the pipeline corresponds to finding a set of paths within the network that all begin at location A and end at location B. This problem, called the **maximum flow** problem, is discussed further in Section 57.8.

The following two examples are best modeled by an undirected graph.

Travel Directions Problem: Given a map showing all highways and their interchanges, with the distance for each segment between two interchanges where both A and B are on a highway, determine the shortest way to get from city A to city B using the US interstate highway system. If we assume all US highways support two-way traffic, and the distance between two interchanges is the same regardless of which direction you are traveling, the data for this problem is best modeled as an undirected graph. The weight for the edge between interchange X and interchange Y would be the number of miles for that stretch of road. Determining the shortest route from city A to city B corresponds to finding the shortest path (based on the sum of the edge weights) from A to B. Although this is an undirected graph, the same algorithms are used as those used for a directed graph.

Optical Network Problem: Given the costs of laying optical fiber between pairs of locations, decide where to lay optical fiber to reach all locations with the shortest total length of fiber. Since two-way communication can be performed using the same infrastructure, the provided data is best modeled as an undirected graph. The weight of the edge between each pair of locations would be the cost of laying the fiber between them. Pairs of locations that cannot be directly connected would have no corresponding edge in the graph. Computing the minimum infrastructure cost corresponds to selecting the subset of edges with the least total weight that provides a path between every possible pair of locations. This problem is the **minimum spanning tree** problem, which is discussed further in Sections 57.3 and 57.4.

Types of Graphs: A Summary

Relationships with an associated value are represented using a **weighted graph** (in which a real-valued weight is associated with each edge). Boolean relationships can be represented using an **unweighted graph**. The data structures and algorithms are fundamentally the same for directed and undirected graphs so we do not distinguish with different ADTs.

While the data structures to implement unweighted and weighted graphs are quite similar, the addition of weights significantly changes the algorithm design. Since the abstract graph classes include these algorithms, we separate the unweighted and weighted graph ADTs. We introduce the Graph ADT (for unweighted graphs) and the WeightedGraph ADT (for weighted graphs).

For both the weighted graph and unweighted graph, the choice of directed or undirected is given as a boolean argument to the constructor. Thus, four different types of graphs result: an **unweighted undirected graph**, an **unweighted directed graph**, a **weighted undirected graph**, and a **weighted directed graph**. The graph of [Figure 2.5](#) is a weighted directed graph where edge weights representing the exchange rates.

One other variation that occurs is a **multigraph** in which there can be multiple edges between the same pair of vertices. For example, if one is storing flight information between cities, there may be several different flights (with different costs, departure times, etc.) from airport A to airport B, and for many applications it would be important to have one edge from vertex A to vertex B for each flight. (See Section 56.1 for a case study based on this application.) As with the distinction between undirected and directed graphs, the data structures and algorithms are quite similar for multigraphs and so no distinction in ADT is needed to accommodate for a multigraph.

Chapter 3

How to Use This Book

This book is designed to help computer science students and application developers select data structures and algorithms that best support the requirements of their applications. The book is organized in a top-down data-centric fashion encouraging the following design process.

1. Think about the way in which the application needs to use information.
2. Select an abstract data type (ADT) appropriate for that use.
3. Select a data structure that efficiently supports the ADT operations most frequently used by the application, perhaps customizing the data structure implementation for the application.

The algorithms covered in this book are not presented in isolation, but instead are presented in the context of the operations of the ADTs and data structures they support.

Particular care has been taken to help readers quickly identify the sections of the book that are most appropriate for requirements of the application at hand. Consequently, making productive use of this book does not require reading it cover to cover. We expect that readers initially will become familiar with the material in Part I, especially Chapter 2 on selecting an ADT. From there, we expect that the use of this book will be application driven. That is, given a particular set of application requirements, the reader will follow the decision process detailed in Chapter 2 and then turn to the appropriate chapters.

The ADT selection process is summarized by the decision tree inside the front cover of this book. The text and vertical placement of the leaves of the tree correspond to the text and vertical placement of tabs that appear in the margins throughout Part II and Part III. Once familiar with the decision process, finding the right section of the book for a particular application is simply a matter of following the decision tree and turning to the corresponding tabbed section.

3.1 Conventions

We use the following conventions throughout this book. Some of these are commonly used conventions, and others are specific to this book.

- Method names use **camelCase**, with all but the first word capitalized. In prose, we also use italics for the method names. For example, the method *isEmpty* returns *true* when there are no elements in a collection.
- The names for constants are written in all upper case letters and italicized.
- Class names use **CamelCase**, with the first letter of each word in the name capitalized. For example, the binary search tree class is named `BinarySearchTree`. We often leave out the word “class” in the prose. For example we might talk about “the `BinarySearchTree` *add* method” when speaking of the *add* method of the `BinarySearchTree` class.

- When referring to an instance of a class, we use a plain font and standard capitalization and spacing. For example, an instance of the `BinarySearchTree` class is generally referred to as a binary search tree.
- Interface names use CamelCase in italics. For example the `BinarySearchTree` class implements the *OrderedCollection* interface. Sometimes, it is necessary to refer to the implementations of an interface or ADT as a group. For example, when we say “*OrderedCollection* data structures,” we refer to the set of all data structures that implement the *OrderedCollection* interface.
- When referring to an ADT that corresponds to an interface, we use CamelCase with a plain font, as in the `OrderedCollection` ADT.
- When referring to an instance of an ADT, we use a plain font, and standard capitalization and spacing. For example, we might talk about an “ordered collection” when referring to an instance of an arbitrary ordered collection ADT.

3.2 Parts II and III Presentation Structure

Most chapters of Part II and Part III are grouped according to the ADT. The first chapter of each group describes the interface for that ADT, followed by a set of chapters that describe particular implementations for that ADT. To set the stage, Chapter 5 provides the foundations used throughout Parts II and III to support good object-oriented design. Some of the material in Chapter 5 reviews standard design techniques such as the use of wrappers and iterators. The rest of Chapter 5 discusses classes and interfaces we have designed to support data structure and algorithm implementations. Of particular note is the **Locator** interface in Section 5.8. The *Locator* interface extends the Java *Iterator* interface with methods that separate navigation from element access. For example, a *get* method provides access to the current element without advancing the locator. Section 5.8.5 describes how locators provide flexibility in cases of concurrent modification, such as when a data structure is mutated during iteration. Rather than invalidating locators for every concurrent modification, we introduce the concept of a **critical mutator**, a method that may invalidate an existing locator to cause a concurrent modification exception if that locator is subsequently used for navigation. Methods that involve only the element at the current position of the locator (e.g., *remove*) can be performed even if the locator has been invalidated. In particular, a **tracker** is a special kind of locator that allows an application to track a particular object even after a critical mutator has been executed.

3.2.1 ADT Chapters

Each chapter that presents an ADT generally includes the following sections.

Case Study: Presents a case study to illustrate an application for which the ADT is well-suited. These case studies are intended to provide further examples of how various data structures and algorithms can be applied. They also exemplify the process by which those particular data structures and algorithms were selected for the application. Therefore, reading the case studies can provide additional insight into this decision process. Application developers contemplating the use of a particular data type can read a related case study before deciding.

Interface: Describes all required methods. Although constructors are not part of a Java interface, we also include a description of the constructors. The information is provided in a format similar to that used by the Java library documentation.

Critical Mutators: Lists the critical mutators of the ADT. The list is maximal. That is, particular implementations of the ADT are free to remove a method from the list if it is possible to implement that method in a way that does not invalidate locators. However, if an interface method is not listed here, then no implementation may provide an implementation that could invalidate a locator. In this way, the user of the ADT is guaranteed that if no listed critical mutators are called while a locator is active, then the locator will continue to operate as expected.

Locator Interface: Describes locator methods that are added to those of the parent class, or for which the semantics add additional constraints from that of the parent class.

Competing ADTs: Lists possible alternative ADTs that should be considered before making a final determination to use the ADT. This section provides a safety net to prevent developers from going astray during the decision process, so that time is focused on the appropriate abstractions.

Terminology: Introduces key definitions used in the discussion of more than one data structure that implements the ADT. Terminology specific to a single data structure appears in its chapter.

Selecting a Data Structure: Guides the reader in selecting among various ADT implementations.

Summary: Provides a brief description of the data structures for each implementation of the ADT, along with a **class hierarchy diagram** showing their relationships. For example, see Figure 9.3. In the class hierarchy diagrams, abstract classes are shown as parallelograms, concrete classes as rectangles, and interfaces as rounded rectangles. Solid lines represent subclass relationships, with the arrow pointing from the child to its parent. (Typically the parent is also placed above its children on the page.) A dashed line from a class to an interface indicates that the class implements that interface.

Comparison Table: Facilitates a quick (“at-a-glance”) comparison between the data structures we present for that ADT. These “at-a-glance” tables typically define four levels of performance (excellent, very good, good, fair) with a different symbol associated with each level. The advantage of providing a relative comparison is that the differences between the ADTs are easy to see. However, so that meaningful comparisons can be made in this form for all ADTs, the reader must be aware that the definition of “excellent,” “very good,” “good,” and “fair” often differ across ADTs. Comparisons should never be made across two different “at-a-glance” comparison tables. For each data structure presented, a table with the asymptotic time complexity is given at the end of the chapter for direct comparisons with other data structures in the book.

Design Notes: Discusses the important implementation techniques, including the use of some classic design patterns that are illustrated in the data structures for this ADT. This portion discusses any design techniques that are broader than any single data structure that implements the ADT. Additional “design notes” in each chapter present implementation techniques that apply to a single implementation of the ADT.

Further Reading: Summarizes related work to give both historical references and references to other good sources on related ADTs and data structures.

3.2.2 Data Structures

Each chapter that covers a specific data structure begins with the package and its ancestors in the class hierarchy. For example, consider the Java class declaration header.

```
public RedBlackTree<E> extends BalancedBinarySearchTree<E>
                        implements OrderedCollection<E>, Tracked<E>
```

Although sufficient for a compiler, the above header does not include the full context in which the class is defined. Therefore, we instead show the following expanded form:

```
AbstractCollection<E> implements Collection<E>
  ↳ AbstractSearchTree<E> implements OrderedCollection<E>
    ↳ BinarySearchTree<E> implements OrderedCollection<E>, Tracked<E>
      ↳ BalancedBinarySearchTree<E> implements OrderedCollection<E>, Tracked<E>
        ↳ RedBlackTree<E> implements OrderedCollection<E>, Tracked<E>
```

In the class hierarchy, the bottom line shows the class being covered in the chapter. The class it extends is immediately above it (and the rest of the inheritance hierarchy is shown as you move upward). All interfaces that are implemented at each level are also shown. Observe that all information needed for the class declaration is included here.

In addition to the class hierarchy, the introductory section for each data structure chapter includes the following components:

Uses: Lists the other data types or algorithms on which this data structure depends. Since it is often necessary to understand these to fully understand the data structure being presented, these dependencies are listed with cross references.

Used By: Lists other data structures and case studies that use this data structure.

Strengths: Summarizes the strengths of this data structure. If these strengths match the way the application will use the data structure, this implementation of the ADT is likely to be the right choice.

Weaknesses: Summarizes the weakness of this data structure. If these are a problem for the application, then a different data structure or different ADT may be more appropriate.

Critical Mutators: Lists the critical mutators for this data structure. This list may not contain an ADT interface method unless it is listed as a critical mutator for the ADT.

Competing Data Structures: Describes alternative data structures that should be considered before making a final decision to use the data structure described in that chapter. These should be read carefully to avoid wasting time and effort on an inappropriate choice.

After the introduction, each data structure chapter contains the following sections:

Internal Representation: Presents the internal representation used by the data structure. It includes the following sections:

Instance Variables and Constants: Explains the purpose of each instance variable and constant defined by the data structure.

Populated Example: Illustrates the data structure by giving an example instance.

Abstraction Function: Defines a function that maps from the internal representation to the user's view of the abstraction.

Terminology: Introduces key definitions used throughout the chapter. For some data structures, this section will precede the abstraction function section.

Design Notes: Discusses any design techniques illustrated in this chapter. Interesting uses of design patterns are briefly discussed, along with a forward reference to the section of Appendix C where that design pattern is discussed in more depth.

Optimizations: Summarizes the optimizations that have been made or could be made to reduce overhead, or that could be made for applications that do not use of all the primary methods of the interface. This section is designed to assist developers in customizing the data structures and algorithms for particular application needs. In general, our implementations tend to favor generality and code reuse over optimizations that do not affect asymptotic complexity.

We also discuss situations in which we choose to optimize the code, and explain the reason for that choice.

Representation Properties: Lists representation invariants and invariants that relate the internal representation to the user-level abstraction, in accordance with the abstraction function. They help the reader understand the internal representation and simplify the discussion of correctness. Before making custom modifications, developers should read and understand the relevant properties and correctness highlights to ensure that the result of the planned modification will continue to satisfy the representation properties.

For each method, there is an informal discussion of correctness based on the representation properties. For accessors, the properties are used to argue why the accessor returns the correct answer. Since the accessors do not change the data structure, all properties are preserved by them. For the constructor and mutators it is important that each representation property is preserved. For the representation properties that are not affected by a mutator, we do not explicitly state that the property is unaffected. Instead, we just argue that each representation property that is affected by the mutator is preserved.

For mutators that add or remove elements from the data structure, there is often the need to relate the old and new values of the variables used within the abstraction function. As a convention, we use the name of the variables with a prime (') to represent their value after the method has executed. For example, most of the data structures have a property that $size = n$ where $size$ is an instance variable, and n denotes the number of elements in the collection. Consider a method that adds one new element to the collection. Then $size' = size + 1$ and $n' = n + 1$. Thus if $size = n$ before the add method executed then it follows that $size' = n'$ and so this property is maintained. In straightforward cases, such as this one, we typically do not give such a formal argument.

Methods: Provides Java code for each constructor and method followed by a discussion of correctness. Since each implementation involves many methods, they are organized in the following standard subsections for easy access.

Constructors: Describes the way in which the instance variables are initialized by the constructors. We group with the constructors any **factory methods** (see Section C.9) like *createNewNode* that allocates an object of the appropriate type.

Trivial Accessors: Describes accessors that return the value of an instance variable (such as *getSize*) and very simple methods (such as *isEmpty*).

Representation Accessors: Describes internal methods that map the internal representation to the user's view of the data or that perform simple structural navigation.

Algorithmic Accessors: Describes accessors that perform a look up function or computation on the data.

Representation Mutators: Describes mutators that modify the internal representation but do not change the user view. In other words, the value of the abstraction function is not changed by those methods.

Content Mutators: Describes mutators that change the data held by the data structure in a way that changes the user view. In other words, these methods usually result in a change to the value of the abstraction function.

Locator Initializers: Describes the methods used to initialize a locator other than when the element is added to the collection.

Locator Class: Describes the inner class that is included with each data structure to mark or track the elements in the data structure. (See Chapter 7 for further discussion.) Recall that the *Locator* interface is an extension of Java's *Iterator* interface.

Performance Analysis: Discusses the time and space complexity for the methods. To reduce unnecessary repetition, methods that are implemented in *AbstractCollection* and take constant time for all implementations are discussed only in Section 8.6 where they are first presented.

Quick Method Reference: For easy access, we provide the signatures and page numbers for all methods (including inherited methods) associated with each class and inner class as an alphabetized list (by the method name) of the signatures for all public methods, followed by an alphabetized list of the signatures for all internal methods. Method names are in a bold font for easy recognition and constructors are shown in italics. To the left of the method signature is the page number where the implementation is provided. A bold font for the page number indicates that the method is declared within in the chapter, whereas a plain font page number refers to inherited code. We have included inherited methods in the list since complete understanding of the data structure implementation may require reviewing some of the inherited methods.

3.2.3 Algorithms

Because this book integrates the presentation of algorithms within the data types that support them, each algorithm presentation provides an illustration of how that data type can be used to solve a computational problem.

For example, within the abstract positional collection class, we present a variety of sorting algorithms (insertion sort, mergesort, heap sort, tree sort, quicksort, radix sort, and bucket sort) and also a selection algorithm that finds the i^{th} smallest element within the collection. By setting $i = \lceil n/2 \rceil$, the median is found. We present both the standard array-based implementations of these algorithms (Section 11.4), as well as direct list-based implementations (Section 15.5). Another unique feature of our presentation of sorting algorithms is that both radix sort and bucket sort allow the user program to provide an object as a “plug-in” to define how the elements are to be divided into digits, or partitioned into buckets. Also, within the sorted array data structure (Chapter 30), we present the binary search algorithm.

In many cases, the algorithms are implemented in terms of the ADT interface and included in an abstract implementation of the ADT. The advantage of such an approach is that the algorithm (in both its presentation and instantiation) is decoupled from the particular ADT implementation. For example, within the abstract graph class we present depth-first search, breadth-first search, topological sort, an algorithm to compute strongly connected components. Within the weighted graph class we present Dijkstra's single-source shortest path algorithm, Floyd-Warshall all-pairs shortest path algorithm, Prim's minimum spanning tree algorithm, Kruskal's minimum spanning tree algorithm, and included a discussion of maximum flow algorithms (including the implementation of the Ford-Fulkerson algorithm).

3.3 Appendices and CD

Appendix A provides a brief overview of the major features of the Java programming language and Appendix B provides an overview of asymptotic notation and complexity analysis. Appendix C provides a catalog of the design patterns that we illustrate with references to the chapters in which they can be found. Finally, all of the code for the data types, including interfaces, implementations, and algorithms is included on the accompanying CD. Documentation, in javadoc format, is also provided. Some JUnit test cases are also provided on the CD to assist in the testing of customized implementations.

Part II

COLLECTION DATA STRUCTURES AND ALGORITHMS

Chapter 4

Part II Organization

[Table 4.1](#) provides a visual summary for the organization of our coverage of ADTs and their corresponding data structures. Note that graphs are covered in Part III. The algorithms we present are integrated within the appropriate chapter. For maximum use, we place algorithms as high as possible in the class hierarchy while still ensuring that we provide an efficient implementation for them.

Chapter 5, presents the foundational classes and interfaces used throughout Part II and Part III. These support the design of data structures and algorithms with locators that are robust to concurrent modifications. They also lay a foundation for good object-oriented design leading to reusable components that are extensible and easily customized.

Next, in Chapter 6, we present the Partition ADT, *PartitionElement* interface, and the union-find data structure. We include a case study showing how to apply the Partition ADT to the problem of efficiently merging two data structures in a way that locators remain valid. The overhead added to support the merge is almost constant.

A significant portion of Part II presents ADTs that maintain a collection of elements. In Chapter 7, we present the *Collection* interface. Within this discussion, we present the *Locator* interface that provides a mechanism by which the user can perform operations via either a marked location in the data structure or tracker that follows a particular element, without exposing the internal representation to the user.

Our presentation of collections starts with the *PositionalCollection* interface, the appropriate choice for manually positioned collections. Chapter 10.1 presents the *AbstractPositionalCollection* class that includes methods used by several of the positional collection data structures. We begin with the array-based implementations. The foundation for these is the array (Chapter 11) data structure, which includes implementations for insertion sort (Section 11.4.1), mergesort (Section 11.4.2), heap sort (Section 11.4.3), tree sort (Section 11.4.4), and quicksort (Section 11.4.5). In Section 11.4.6, radix sort is presented in terms of a user provided digitizer (see Section 5.3) so that it can be applied for any data type for which a digitizer can be defined. This generality gives applications the flexibility to define how each element is divided into digits to yield optimal performance for the particular application. Similarly, in Section 11.4.7, we present the bucket sort algorithm where the application provides a bucketizer (see Section 5.4). Finally, Section 15.6 presents a randomized selection algorithm to find the i^{th} smallest element in a collection in linear time. By setting $i = \lfloor (n - 1)/2 \rfloor$, where n is the number of elements in the collection, this algorithm computes the median. These algorithms run efficiently on the other array-based implementations (circular array, dynamic array, dynamic circular array, and tracked array), which are presented in Chapters 12–14.

Chapter 15 presents the singly linked list data structure, the first of the two list-based implementations we provide for the positional collection. Within this chapter we provide efficient list-based implementations for the sorting algorithms presented in Section 11.4. In particular, we present efficient in-place linked list implementations for insertion sort, mergesort, quicksort, radix sort bucket sort, and the selection (median finding) algorithm. List-based implementations of heap sort and tree sort are also provided. Chapter 16 extends the singly linked list to build a doubly linked list. This chapter illustrates how good object-oriented design allows most functionality to be inherited. Few

methods need to be modified to maintain the references to the previous element in the collection, and to make use of these references to improve efficiency when appropriate.

Chapters 17–19 provide implementations for three specializations of the *PositionalCollection* interface: the Buffer ADT, Queue ADT, and Stack ADT. We also use these data structures to illustrate an implementation of a **bounded data structure**, where a user-provided bound on the size of the collection can be provided. If desired, a similar approach could be used for any other data structure for which it is desirable to maintain a bound on the size.

The remaining ADTs that implement the *Collection* interface are algorithmically positioned. As discussed on page 25, these data structures can be untagged or tagged. While the underlying data structures are very similar for the corresponding untagged and tagged ADTs, the interfaces differ. We first present the data structures for the untagged ADTs and then extend these for the tagged ADTs. The most basic untagged collection ADT is the Set ADT that maintains a collection of elements in which no two elements are equivalent. Chapters 20–23 present the *Set* interface and data structures. In Chapters 24–48, we present the untagged collection ADTs, for which the elements are comparable and the collection can hold equivalent elements.

Chapters 24–28 present the *PriorityQueue* interface and data structures that implement it. The *OrderedCollection* interface and data structures are covered in Chapters 29–38. Related algorithms are presented in the context of these implementations. For example, in Chapter 30, as part of our coverage of the sorted array data structure, we present the binary search algorithm for efficiently searching for a desired element in a sorted array (Section 30.3.3). The *OrderedCollection* ADT has so many different implementations that it is not possible to present all of them in depth. Chapters 29–38 cover nine of the most common implementations. Chapters 39–45 present the *Digitized-OrderedCollection* interface and data structures, and Chapters 46–48 present the *SpatialCollection* interface and data structures.

After presenting all the untagged algorithmically positioned collection ADTs, we move to the tagged algorithmically positioned collection ADTs, in which the user provides a tag for each element, and this tag is used as the basis for the logical organization of the data. While the user view and interface in a tagged collection is different from the untagged collection, the underlying data structures are very similar. Ungrouped tagged collections are covered in Chapter 49 and grouped tagged collections are covered in Chapter 50.

In a tagged collection, each element and its associated tag is encapsulated as a **tagged element**. Section 49.1 presents the *TaggedElement* class. Section 49.2 presents the *TaggedCollection* interface. Section 49.4 briefly discusses other ADTs that may be appropriate in a situation when a tagged collection is being considered. Guidance in selecting a tagged collection ADT is provided in Section 49.5. Each tagged collection data structure is implemented as a wrapper for any collection implementation, where each element in the collection is a tagged element. Section 49.6 presents the tagged collection wrapper that allows any collection data structure to be wrapped. Section 49.7 describes the interface for the Mapping ADT, and outlines the process of selecting the appropriate set data structure to wrap. A sample implementation for the *OpenAddressingMapping* data structure, that simply includes the desired constructors, is also provided. Section 49.8 describes the interface for the *TaggedPriorityQueue* ADT, and overviews the process of selecting the appropriate priority queue data structure to wrap. This section includes the *TaggedPriorityQueueWrapper* that extends the *TaggedCollectionWrapper* to support the additional methods of the *TaggedPriorityQueue* interface, and also a sample implementation for the *TaggedPairingHeap*. The remaining sections in Chapter 49 use the same approach to present the *TaggedOrderedCollection* (Section 49.9), *Tagged-DigitizedOrderedCollection* (Section 49.10), and *TaggedSpatialCollection* (Section 49.11) ADTs and data structures.

While many applications are best supported by associating a single element with each tag, some applications are best supported by associating a collection of elements with each tag. Chapter 50 presents the tagged bucket collections, each of which wraps the corresponding tagged collection. Sections 50.1 and 50.2 present two cases studies for applications in which a tagged bucket col-

lection is a good ADT choice. A bucket factory is used to create a new bucket for each newly added tag. Section 50.3 presents the *BucketFactory* interface. Using the **abstract factory** design pattern (Section C.1) provides the flexibility to create bucket collections having any bucket type. Section 50.4 presents the *TaggedBucketCollection* interface, in which all elements with equivalent tags are stored together in a bucket associated with that tag. Guidance in selecting a tagged bucket collection ADT is provided in Section 49.5. Section 50.7 presents the tagged bucket collection wrapper that allows any tagged collection data structure to be wrapped, and any bucket factory to be used to create the buckets. Finally, Section 50.6 overviews the process of selecting a data structure for the *BucketMapping*, *TaggedBucketPriorityQueue*, *TaggedBucketOrderedCollection*, *TaggedBucketDigitizedOrderedCollection*, and *TaggedBucketSpatialCollection* ADTs.

Partition ADT and the Union Find Data Structure					Ch. 6				
Collection of Elements	Collection Interface and Locator Interface				Ch. 7				
	Abstract Collection				Ch. 8				
	Manually Positioned		PositionalCollection Interface		Ch. 9				
			AbstractPositionalCollection		Ch. 10				
			Data Structures for General PositionalCollection ADT (access at any position)		Array	Ch. 11			
					CircularArray	Ch. 12			
					DynamicArray and DynamicCircularArray	Ch. 13			
					TrackedArray	Ch. 14			
					SinglyLinkedList	Ch. 15			
					DoublyLinkedList	Ch. 16			
			Specialized Positional Collection ADTs (access only at ends)		Buffer	Ch. 17			
	Queue	Ch. 18							
	Stack	Ch. 19							
	Algorithmically Positioned		Set ADT (access only for determining membership, elements are unique)		Set Interface	Ch. 20			
					DirectAddressing	Ch. 21			
					SeparateChaining	Ch. 22			
					OpenAddressing	Ch. 23			
			Elements are Comparable (need not be unique)		PriorityQueue ADT (access elements with highest priority)		PriorityQueue Interface	Ch. 24	
							BinaryHeap	Ch. 25	
							LeftistHeap	Ch. 26	
							PairingHeap	Ch. 27	
							FibonacciHeap	Ch. 28	
					OrderedCollection ADT (access based on an ordering of elements)		OrderedCollection Interface		Ch. 29
							SortedArray	Ch. 30	
							AbstractSearchTree	Ch. 31	
							BinarySearchTree	Ch. 32	
							BalancedBinarySearchTree	Ch. 33	
							RedBlackTree	Ch. 34	
							SplayTree	Ch. 35	
							BTree	Ch. 36	
DigitizedOrderedCollection ADT (access elements using prefix relations)					BPlusTree	Ch. 37			
					SkipList	Ch. 38			
					DigitizedOrdered Collection Interface		Ch. 39		
		Trie Node Types	Ch. 40						
SpatialCollection ADT (access based on multiple orderings of the elements)		Trie	Ch. 41						
		CompactTrie	Ch. 42						
		CompressedTrie	Ch. 43						
		PatriciaTrie	Ch. 44						
Tagged		Ungrouped		TernaryTrie	Ch. 45				
				SpatialCollection Interface	Ch. 46				
		KDTree	Ch. 47						
Grouped		QuadTree	Ch. 48						
		TaggedCollection ADTs		Ch. 49					
Tagged		Grouped		TaggedBucketCollection ADTs		Ch. 50			

Table 4.1 An overview of the ADTs and data structures presented in Part II of this book.

Chapter 5

Foundations

Successful object-oriented design involves reusable components and frameworks that support extensible, easily maintained software systems. In this chapter we present the foundations used throughout this book to support these goals. A reader interested in learning about object-oriented design might choose to read this chapter in its entirety. However, a reader using this book to find the most appropriate data structure or algorithm for a particular application, would be best served by reading Sections 5.1, 5.2, 5.6, 5.7, 5.8, and 5.10, and refer to the other sections as needed.

Section 5.1 discusses wrappers, a design pattern used throughout this book when one class provides much of the functionality needed by another, yet they do not share the same interface. Section 5.2 presents support for maximizing the generality and applicability of our implementations. This includes the handling of *null* as a valid data element (Section 5.2.1), general approaches for testing objects for equivalence (Section 5.2.2), and comparing objects (Section 5.2.3).

Some algorithms and data structures view each object as a sequence of digits. Section 5.3 describes the *Digitizer* interface that we define and provides a sample implementation for it. Within this book the digitizer is used by radix sort (Section 11.4.6) and the data structures for the Digitized-OrderedCollection ADT (Chapter 39).

In Section 5.4 we define the *Bucketizer* interface that provides a mechanism to distribute elements from a real-valued domain among a discrete set of buckets. We provide a default implementation for this interface. In this book, this interface is used only for the bucket sort algorithm (Section 11.4.7), but provides a general-purpose mechanism that could be useful for other applications.

As discussed in Section A.4, Java is a garbage collected language, meaning that objects persist on the heap until they are no longer reachable from the executing program. A disadvantage of garbage collected languages is the time overhead to perform garbage collection. If a Java program creates many objects for temporary use, the garbage collector may need to run frequently to reclaim the space for new objects. Section 5.5 presents a general technique to reduce this overhead by creating a pool of reusable objects. This mechanism is illustrated in the context of the trie data structure (Chapter 41) and the Patricia trie data structure (Chapter 44). Section 5.6 briefly introduces Java's mechanism to handle concurrency.

As discussed in Section A.10.6, an iterator provides support for moving through a data structure efficiently, without exposing the internal representation. Section 5.7 discusses the drawbacks of the Java iterator, particularly in regards the lack of robustness when the data structure is modified through any means other than the iterator itself. Unlike Java's provided iterator implementations, our Locators (Section 5.8), which extends Java's *Iterator*, are resilient to concurrent modification of data structures. We introduce two types of locators in Section 2.5: **markers** that mark a position within a structure, and **trackers** that keep track of an element in a collection even if the collection is reorganized. Trackers are crucial for efficient implementations of even some standard algorithms, such as the use of a priority queue to implement Dijkstra's shortest path algorithm.

Section 5.8.4 describes how iteration is performed using a locator. Section 5.8.5 describes a mechanism to efficiently determine when a concurrent modification has occurred, and Section 5.9 describes the Version class that supports this mechanism.

While iterators provide the ability to traverse a collection without exposing the internal representation, sometimes the most natural method of iteration is through a recursive method. In these cases, a **visitor** can be used to support traversal in a way that also does not expose the internal representation. Section 5.10 describes visitors and how they are used for traversing data structures. Section 8.5 in the `AbstractCollection` class provides a general-purpose (but inefficient) `VisitingIterator` thread that can be used by a data structure to support iteration without defining a custom iterator class when only *visit*, *hasNext*, and *next* must be supported.

5.1 Wrappers for Delegation

We use the term **wrapper** to refer to an object that encapsulates a reference to another object, and that implements many of its own methods by making calls on the wrapped object. In some cases, the method headers may be identical, and the wrapper simply **delegates** the work by passing the same parameters through to the corresponding method of the wrapped object. Other methods in the wrapper may perform additional work before or after the call to the wrapped object, or may compute different parameter values to be passed to the wrapped object. Finally, some methods of the wrapper may provide additional functionality and may not use the wrapped object at all. Many design patterns, such as the adaptor (Appendix C.2) and builder (Appendix C.4), make use of wrappers.

The wrapped object may be created within the constructor of the wrapper, or may be passed to that constructor. The latter case must be handled carefully, since the internal representation is not fully encapsulated. If code that calls the wrapper constructor retains a reference to the wrapped object, it could make further (possibly dangerous) modifications of that object without calling methods on the wrapper. However, the advantage of passing in the wrapped object is that it offers the user some control over the specific type of object being wrapped. For example, if the wrapper can wrap any ordered collection that is provided to the constructor, then the user can select the ordered collection data structure that is preferred for that application.

The difference between extending a class and wrapping an object of the same type is subtle, but important. In both cases, the result is a new class that has somewhat different behavior from the original. Also, a wrapper may result in more code than the inherited class, because each method to be delegated must appear in the wrapper class as well. However, wrappers provide the advantage of being able to hide functionality of the wrapped object that is not desirable for the wrapper type. For example, our `Stack` (Chapter 19) wrapper with last-in first-out semantics wraps the `dynamic array` (Chapter 13) class, hiding methods of the `dynamic array` class that would allow addition or removal from the middle of the data structure. Such hiding would not be permitted if `Stack` extended `DynamicArray`, due to the type safety and polymorphism, as described in Section A.10.5.

The following **`NonMutatingIterator`** wraps an iterator to prevent users from calling the *remove* method in cases where allowing such mutations could violate a representation invariant. Users of a non-mutating iterator can iterate over the collection, but cannot use the iterator to modify the collection.

```
public class NonMutatingIterator<T> implements Iterator<T> {
    private Iterator<? extends T> wrapped;
    public NonMutatingIterator(Iterator<? extends T> iterator) {
        this.wrapped = iterator;
    }
    public boolean hasNext() { return wrapped.hasNext(); }
```

```

public T next() { return wrapped.next(); }
public void remove() { throw new UnsupportedOperationException(); }
}

```

Since a wrapper must make an extra method call, there is a very small increase in time complexity as compared to repeating the code from the wrapped class. However, advantages of reusing code usually outweigh any concerns about this minimal extra cost.

5.2 Objects Abstract Class

In this section we describe various utilities defined in Objects abstract class that support many of the data structures presented within this book. These general-purpose items handle problems like how to mark empty and deleted elements, comparing objects for equivalence, and comparing objects to order them.

5.2.1 Singleton Classes: Empty and Deleted

In many data structure implementations, there is an implicit assumption that *null* is not a valid data element. However, to provide the most general purpose implementations possible, we assume throughout this book that *null* might be an element in a collection or the data associated with some element. The problem then arises of distinguishing between the cases of a *null* element and no element at all.

The constant *EMPTY* is used for any implementation in which a user defined element is directly referenced by a (sparse) array.

```

public static final Object EMPTY = new EmptySlot();           //empty sentinel
static class EmptySlot{}

```

Similarly, for some data structures (such as open addressing, Chapter 22), a separate object is needed to mark the array slots that previously held an element that has since been deleted. The singleton constant *DELETED* is used to mark these slots. Using a value of a different type ensures that no element in the collection is equivalent to this value without using any significant amount of storage.

```

public static final Object DELETED = new DeletedElement();   //deleted sentinel
static class DeletedElement{}

```

The above definitions are example uses of the singleton design pattern (see Section C.14), which ensures that each class has one instance and provides a global access point. We use this design pattern in many places to reduce space.

5.2.2 Object Equivalence

Many data structures have methods that need to determine whether two objects are considered equivalent. For example, a set abstraction may prohibit duplicate elements and may therefore need to determine if an element being added is equivalent to any element already in the set. For objects *o1* and *o2*, if *o1* == *o2*, then the two object references are equal and *o1* and *o2* refer to the same object, so they are equivalent. However, sometimes *o1* and *o2* do not refer to the same object, yet *o1* and *o2* should be considered equivalent. For example, it could be that *o1* and *o2* are references of two objects of the same type that hold the same values in their instance variables. As another example, *o1* and *o2* could be instances of a rational number class where *o1* holds $\frac{4}{3}$ and *o2* holds $1\frac{1}{3}$.

There are two possibilities for determining equivalence: either the objects themselves determine whether they are equivalent to other objects, or the application provides its own external criteria for making that determination. When the objects themselves define equivalence, the standard practice in Java is for their classes to override the *equals* method that is inherited from the class *Object*. The method call *o1.equals(o2)*, returns *true* if and only if *o1* considers itself to be equal to *o2*. The default implementation returns *true* only if *o1 == o2*. When a user-defined class overrides *equals*, Java specifies that the result should be **reflexive** (*o1.equals(o1)*), **symmetric** (*o1.equals(o2)* if and only if *o2.equals(o1)*), and **transitive** (if *o1.equals(o2)* and *o2.equals(o3)* then *o1.equals(o3)*). However, Java's *equals* method does not handle the situation when *o1* or *o2* are *null*, a common situation in data structure implementations that allow *null* data values. Therefore, we provide a static method (in the abstract class **Objects**) that can be used to determine if two (possibly *null*) object references are equivalent.

equivalent(Object o1, Object o2): Returns *true* if and only if *o1* and *o2* are equivalent. This method assumes the *equals* method is symmetric. If that were not the case then *o1.equals(o2)* would need to be replaced by *o1.equals(o2) && o2.equals(o1)*, which would make this method less efficient.

```
public static final boolean equivalent(Object o1, Object o2) {
    return o1 == o2 || (o1 != null && o1.equals(o2));
}
```

While the *equivalent* method allows an application programmer to define a notion of equivalence for a given data type, often two different applications using the same data type (e.g., a two-dimensional point) desire a different notion of equivalence. When an application wishes to define its own notion of equivalence between elements, it needs a mechanism by which it can supply that definition to the data structure in question. For example, an application may want to say that two objects should be considered equivalent if a particular field matches, even though the *equals* method for those objects uses different criteria. For example, perhaps points p_1 and p_2 are considered equal if their *x*-coordinates and *y*-coordinates are equal, yet some application prefers to consider two points to be equivalent if they are the same distance from the origin.

In Java, the standard practice for application-specific comparison is for the application to provide an object implementing the *java.util.Comparator* interface with a *compareTo* method that takes any two objects to be compared and returns the integer value zero if and only if it considers the two objects to be equivalent. Comparators offer flexibility: applications can compare the same objects in different ways by providing different comparators.

Our data structures allow an application-specific definition of equivalence to be provided via a comparator, which is passed to the constructor of the data structure. If no comparator is provided, the following default equivalence tester is used.

```
public static final Comparator<Object> DEFAULT_EQUIVALENCE_TESTER =
    new DefaultEquivalenceTester<Object>();
```

```
public static class DefaultEquivalenceTester<E> implements Comparator<E> {
    public int compare(E o1, E o2) {
        if (Objects.equivalent(o1, o2))
            return 0;
        else
            return -1;
    }
}
```

5.2.3 Object Comparison

We say that an ADT is **comparison-based** if it depends upon there being a well-defined ordering among its elements. As with object equivalence, there are two ways in which the ordering for a comparison-based ADT can be defined. Either the elements themselves define the ordering, or the application provides its own ordering. If the elements themselves define the ordering, then their classes must implement the *Comparable* interface. The *Comparable* interface contains a single method, *compareTo*, that takes another object as a parameter and returns a negative, zero, or positive value depending upon whether the target considers itself to be less than, equal to, or greater than the parameter. If the application defines its own ordering, the objects need not implement the *Comparable* interface. In that case, the application passes a comparator object as an argument to the constructor of the data structure.

When the application does not provide a comparator, we use the following default comparator. When the objects are *Comparable*, it makes the comparison using their *compareTo* methods. Otherwise, a *NotComparableException* (an unchecked exception) is thrown.

```
public static final Comparator<Object> DEFAULT_COMPARATOR =
    new DefaultComparator<Object>();

static class DefaultComparator<E> implements Comparator<E>{

    public int compare(E o1, E o2) {
        try {
            return ((Comparable<E>) o1).compareTo(o2);
        } catch (ClassCastException cce) {
            throw new NotComparableException("Can't compare " + o1 + " and " + o2);
        }
    }
}
```

Throughout the book, we say that objects are **Comparable** (with a capital C) if they implement the *Comparable* interface, and we say that objects are **comparable** (with a lower case c) if either they are *Comparable* or the application defines a comparator that is applicable to those objects.

In some situations, it is desirable to define a comparator that reverses the ordering among the elements. For example, the *PriorityQueue* ADT (Chapter 24) supports efficiently locating and removing the largest (highest priority) element, but it is expensive to find the smallest (lowest priority) element. Suppose an application needs efficient support for efficiently finding and removing the smallest element. For such situations, we define a **reverse comparator** that negates the value returned by the *compare* method. Heap sort 11.4.3 is one example usage of this comparator.

```
public class ReverseComparator<T> implements Comparator<T> {
    Comparator<? super T> comp;

    public ReverseComparator(Comparator<? super T> comp) {this.comp = comp;}

    public final int compare(T a, T b) {return -comp.compare(a, b);}
}
```


5.3 Digitizer Interface

package collection

Digitizer<*T*>

Many data structures and algorithms apply only to data that takes on certain forms. Some sorting algorithms (such as radix sort) and some data structures (such as tries) assume that the elements in the collection can be viewed as a sequence of digits. For example, the base-10 number “562” has a 5 in the 100s place, a 6 in the 10s place, and a 2 in the 1s place. Similarly the initials “ABC” can be viewed as three “digits” where the most significant digit is an “A,” the next most significant digit is a “B,” and the least significant digit is a “C.” The *Digitizer* interface provides support for such applications by allowing an application to define how elements are divided into digits and how those digits are ordered.

More specifically the *Digitizer* interface provides a mechanism for any algorithm or data structure to treat each element in a collection as a sequence of digits where each digit is mapped to an integer from 0 to $b - 1$ where b is the base of the digit. The *Digitizer* interface includes the following methods.

int getBase(): Returns the base. Observe that a base b digit takes on values from 0 to $b - 1$.

int numDigits(*T x*): Returns the number of digits in the element x .

int getDigit(*T x*, int place): Returns the value of digit $place$ for element x .

Indexed Number Class

The following IndexedNumber class illustrates a way to provide access to the digits of a number. We then define a NumberDigitizer class that implements the *Digitizer* interface for indexed numbers. The IndexedNumber class provides constructors for a variety of number types, including the BigInteger class defined in the *java.math* package. Internally, each indexed number is represented as a character array.

```
private char[] digits;
```

Each of the constructors stores the provided integer or String of digits as a character string.

```
public IndexedNumber(int value) {this(""+value);}
public IndexedNumber(long value) {this(""+value);}
public IndexedNumber(BigInteger value) {this(""+value);}
private IndexedNumber(String rep) {digits = rep.toCharArray();}
```

The *numDigits* method returns the number of digits in the element.

```
public int numDigits() {
    return digits.length;
}
```

The *isPrefixFree* method returns *false* since this is not a prefix free digitizer.

```

public boolean isPrefixFree() {
    return false;
}

```

The *ToString* method returns the indexed number represented as a character string.

```

public String toString() {
    return new String(digits);
}

```

The *getDigit* method extracts the character for the given *place* and then subtracts the value of the character 0 to get the integer representation for the digit. This method treats each element as if it was padded infinitely to the left with zeros. For example, 562 can be viewed as ...000562.

```

public int getDigit(int place) {
    if (place ≥ digits.length)                                //pad infinitely to left with 0s
        return 0;
    else if (place < 0)
        throw new IllegalArgumentException("negative place " + place);
    return digits[digits.length - 1 - place] - '0';
}

```

Sample Implementation for the Digitizer Interface

This digitizer works in base 10, and uses standard place value to report the value for each digit.

```

public static class NumberDigitizer implements Digitizer<IndexedNumber> {

    public int getBase() {
        return 10;
    }

    public int numDigits(IndexedNumber x) {
        return x.numDigits();
    }

    public boolean isPrefixFree() {
        return false;
    }

    public int getDigit(IndexedNumber x, int place) {
        return x.getDigit(place);
    }

}

```

5.4 Bucketizer Interface

package collection

```
Comparator<T>
└─ Bucketizer<T>
```

Some applications benefit from the ability to partition all possible elements in a collection into a set of groups (or buckets). However, unlike the digitizer which implicitly defines the total ordering of the elements, the bucketizer acts as a comparator for ordering elements within each bucket. A combination of the ordering implicitly defined between the buckets and the comparator used within the buckets defines a total order over the elements. The *Bucketizer* Interface used in conjunction with the *Quantizer* Interface and *Interval* Interface support these.

Describing algorithms and data structures in terms of these interfaces allow each application to customize its use of those algorithms through the particular way in which it implements the interfaces.

Any class that implements the *Bucketizer* interface must include the following two methods, in addition to the *compare* method defined in the *Comparator* interface.

int getNumBuckets(): Returns the number of buckets used.

int getBucket(T x): Returns the bucket to which element *x* belongs. The value returned must be an integer in the range 0 to *getNumBuckets()* – 1.

See Section 11.4.7 for an example use for the *Bucketizer* interface.

Quantizer Interface

The *Quantizer* interface allows an application program to provide a mechanism to convert an element *x* in the collection to a double. (While an integer could have been used instead of a double, using a double provides the maximum flexibility.)

```
public interface Quantizer<T> {
    public double getDouble(T x);           //converts x to a double
}
```

Interval Interface

The *Interval* interface provides a way for an application program to provide a minimum and maximum element when it is known. As illustrated below, when such information is not known, it can be computed during a linear time pass over the collection using a provided *Quantizer*. However, if the range is computed, it is important to note that if the minimum or maximum values subsequently change, the previous result of the bucketizer would be invalidated.

```

public interface Interval<T> {
    public T getMin();                //returns min element in the range of a collection
    public T getMax();              //returns max element in the range of a collection
}

```

DefaultBucketizer<T> implements Bucketizer<T>

As an illustration, we now describe a default implementation for the bucketizer that can be used for a variety of applications. In Section 11.4.7, we use this bucketizer in an implementation of bucket sort.

The bucketizer uses the following simple *Quantizer* interface and implementation. It has five instance variables:

```

Collection<T> coll;                //collection on which it is defined
Quantizer<? super T> quantizer;    //quantizer provided to convert element to double
double min, max;                  //minimum and maximum quantized value in collection
double range;                     //the difference of the maximum and minimum values
int numBuckets;                   //the number of buckets

```

The constructor takes *coll*, the collection to which the bucketizer applies and *quantizer*, for converting each element to a double. The provided implementation sets the number of buckets to be equal to the number of elements in the collection.

```

public DefaultBucketizer(Collection<T> coll, Quantizer<? super T> quantizer) {
    this.coll = coll;
    this.quantizer = quantizer;
    numBuckets = coll.getSize();
    computeRange();
}

```

Recall the *compare* method takes *o1*, one element to compare and *o2*, the second element to compare. It returns a negative integer when *o1* is less than *o2*, zero when *o1* is equivalent to *o2*, and a positive integer when *o1* is greater than *o2*.

```

public int compare(T o1, T o2) {
    double double1 = quantizer.getDouble(o1);
    double double2 = quantizer.getDouble(o2);
    if (double1 < double2)
        return -1;
    else if (double1 == double2)
        return 0;
    else
        return 1;
}

```

To determine the correct bucket for an element, the range of values created by the quantizer must be known. If a collection maintains its minimum and maximum values, it can implement the *Interval* interface for the bucketizer to efficiently access those values. Otherwise, a linear time pass through the collection is used to compute the minimum and maximum double which the quantizer

returns. This should be done only after the minimum and maximum values have been added to the collection, so a default bucketizer should be created only for a collection that implements *Interval* or has already been populated with minimum and maximum elements.

```
protected void computeRange() {
    if (coll instanceof Interval) {                                //min and max available
        min = quantizer.getDouble(((Interval<T>) coll).getMin());
        max = quantizer.getDouble(((Interval<T>) coll).getMax());
    } else {                                                        //compute min and max
        for (T element: coll){
            double x = quantizer.getDouble(element);
            if (x < min)
                min = x;
            else if (x > max)
                max = x;
        }
    }
    range = max - min;                                            //range of quantized values
}
```

The *getNumBuckets* method returns the number of buckets among which the elements are placed.

```
public int getNumBuckets() {
    return numBuckets;
}
```

Finally, the *getBucket* method takes *element*, the desired element in the collection. It returns the bucket where *element* should be placed. The return value is guaranteed to be an integer between 0 and *numBuckets* - 1.

```
public int getBucket(T element) {
    double x = quantizer.getDouble(element);
    if (x == max)
        return numBuckets - 1;
    else
        return (int) (numBuckets * (x - min)/range);
}
```

We close by giving a simple example definition of a *Bucketizer* for the *Integer* class where *c* is a collection of integers.

```
Bucketizer<Integer> b = new DefaultBucketizer<Integer>(c,
    new Quantizer<Integer>() {
        public double getDouble(Integer x) {
            return x;
        }
    });
```

5.5 Object Pool to Reduce Garbage Collection Overhead

Since Java is a garbage collected language, memory is reclaimed only by the garbage collector. Garbage collected languages prevent **dangling references** that are created when a program prematurely frees an object to which it is still holding a reference. The memory allocator may reuse the space that was occupied by the freed object, with potentially disastrous results if the dangling reference is subsequently used.

The disadvantage of garbage collected languages is the overhead of performing garbage collection. If a Java program creates many objects for temporary use, the garbage collector may need to run frequently to reclaim the space for new objects. One way to reduce this overhead is to create a pool of reusable objects. When an object is needed, the pool can provide an existing object (if one is available), or create a new one. When the program is finished using an object, it can return the object to the pool. This functionality is provided by the following abstract Pool class. To use this class, one would create a subclass of Pool and implement the abstract method that creates objects on demand. Our implementation keeps the reusable object in a stack (Chapter 19).

```
public abstract class Pool<T> {
    Stack<T> available = new Stack<T>();
```

The method *allocate* returns an available object from the pool, or creates one if the pool is empty.

```
    public T allocate() {
        if (available.isEmpty())
            return create();
        else
            return available.pop();
    }
```

The method *release* takes *x*, an object that is no longer needed, and moves the given object to the pool of available objects for reuse. Because the object may be reused, we require that the user does not retain a reference to the object released.

```
    public void release(T x) {
        available.push(x);
    }
```

Each *Pool* implementation must provide an implementation of the *create* method that returns a newly allocated object of type T. This method is called when an object is requested from an empty pool.

```
    protected abstract T create();
}
```

We emphasize that when returning an object to the pool, the application program must ensure that no other references to that object will be retained. Otherwise, undesirable object sharing may result. Thus, the use of the object pool is a trade-off that removes the overhead of unnecessary garbage collection at the cost of reintroducing the problem of dangling (type-safe) references. Also, if many objects are in the pool and never used, they will occupy heap memory needlessly, so it may be desirable to place a cap on the size of the pool, or add a method to shrink or empty the pool.

5.6 Concurrency

A **thread** is a flow of execution within a program. All Java programs have at least one thread, known as the **main thread**, which begins executing at the public static *main* method of the program. Programs with a graphical user interface (GUI) have another thread, known as the **event loop**, that processes user events. Applications may create and start additional threads. For example, a thread might execute a periodic task, perform animation, or handle one of many concurrent clients. Each thread has its own execution stack to store the parameters and local variables of the method invocations that occur within that thread. However, all threads within a process share the same heap. Therefore, threads may communicate with each other by modifying and observing shared objects.

Java provides a `Thread` class that is used to create and start threads. To create a thread, one can create an instance of a class that extends the `Thread` class and overrides the *run* method to carry out the activities of that thread. Alternatively, one can create an instance of a class that implements the *Runnable* interface, and pass that instance to the `Thread` constructor. The *Runnable* interface has a single method, *run*, which is used as the starting point of the thread. To start a thread, one simply calls the *start* method on the thread. To provide the illusion that a program's threads run simultaneously, the steps of each in the execution of threads are interleaved with one another. The Java **thread scheduler** controls when each thread gets a chance to execute.

The programmer cannot fully control the decisions made by the thread scheduler, but Java does provide some mechanisms by which the programmer can place some restrictions on which threads may execute concurrently. Broadly, mechanisms and algorithms that restrict the schedule of concurrent activities are known as **concurrency control**. In Java, the primary mechanism for concurrency control is **locking**, which is accomplished using the keyword **synchronized**. For example, in the following code fragment, a lock on *myListOfStrings* is acquired and held throughout the execution of the block. Obtaining the lock prevents another thread from acquiring a lock on the list while the iteration is in progress.

```
synchronized(myListOfStrings) {
    //a lock on myListOfStrings is held throughout this block
    Iterator it = myListOfStrings.iterator();
    while (it.hasNext()) {
        String s = (String) it.next();
        //process object s
    }
}
```

Similarly, a method may be declared as **synchronized**, as shown below. When a thread calls a synchronized method on an object, the thread acquires the lock on that object. The lock is released when the method terminates. (If a static method is synchronized, there is no target object, so the class is locked instead.)

```
synchronized void myMethod() {
    //a lock is acquired on entry and released on exit
}
```

Only one thread at a time may hold a particular lock. Therefore, while a lock is held, any other thread that attempts to acquire that lock will be **blocked** until the other thread releases the lock. Incorrect use of synchronized code can result in **deadlock**, a situation in which each blocked thread holds at least one lock but cannot continue until acquiring a lock held by one of the other blocked threads. When deadlock occurs, none of the threads can continue normal execution (unless one or

more of the threads involved in the deadlock is aborted). In general, testing cannot guarantee the absence of deadlock, since the choices made by the thread scheduler affect the order in which locks are acquired and released. Therefore, one must reason carefully about thread synchronization within a program to ensure that deadlock cannot occur. When synchronization requirements are modest, some simple measures can ensure that deadlock will not occur. For example, if a program is written so that a thread never holds more than one lock at a time, then that thread can never be involved in a deadlock.

Concurrency has important implications for the correct use of data structures. Section 1.3 described a methodology that uses induction to reason about program correctness. Namely, if the constructors establish the truth of a property and each method preserves that property, then the property will hold before and after the execution of each method. Note, however, that the property may not hold *during* method execution. In fact, for any property involving more than one variable, any new values of those variables must be assigned in some order, so the property is likely to be false between those assignments. This is not a problem if only one thread accesses that object, but if there are multiple concurrent threads sharing that object, then one of them might observe the state of the object while another thread is in the middle of executing a mutating method. In such a case, the program could exhibit arbitrarily bad behavior by misinterpreting or corrupting the state of the object. Therefore, it is important to use proper thread synchronization whenever a data structure is shared by concurrent threads.

With few exceptions, the data structures in the book are presented assuming a single thread of execution. If multiple threads share data structures, we assume that steps have been taken to ensure proper synchronization of those threads. It must be guaranteed that no two threads execute simultaneously within methods of the data structure in such a way that would permit observation of an inconsistent state. Otherwise, arbitrarily incorrect behavior could result.

A brute force approach to this problem is to build a **synchronized data structure**, in which all methods are marked as synchronized so Java's locking mechanism prevents two different threads from executing within methods of the data structure simultaneously. To create a synchronized data structure from those in this book, one could directly mark all of the methods for a class as synchronized, or one could place the data structure in a wrapper whose synchronized methods delegate the actual work to the wrapped data structure, as discussed in Section 5.1. This would ensure that a lock is acquired for every method call, thereby preventing one thread from seeing the state of the data structure while another is modifying it.

Note, however, that making every method call synchronized still would not prevent the use of multiple iterators by multiple concurrent threads whose steps are interleaved. To solve this problem, one could place a synchronized block around every use of the data structure. This solution would permit locking a data structure for the entire execution of a loop using an iterator, but has the distinct disadvantage that if a programmer forgets to place a synchronized block around any code using that data structure, then that use would be unsafe. Consequently, this solution is practical only if the use of the shared object can be localized to a small section of the program. An alternative would be to synchronize all methods and use visitors (Section C.17) instead of iterators for traversing the data structures. This would solve the problem since all uses of the visitor would occur within a single (synchronized) method invocation on the data structure and could therefore not be interleaved with the steps of other threads within that data structure.

Alternative approaches to thread-safe data structures are discussed elsewhere [101] and are beyond the scope of this book. In general, however, one should be aware that locking for thread synchronization can take its toll on performance, and, more importantly, that locking must be used carefully to avoid deadlock. For one advanced (and unusual) use of threads, see the `VisitingIterator` class (Section 8.5).

5.7 Iterators for Traversing Data Structures

As discussed in Section A.10.6, an iterator provides a solution to the problem of efficient traversal of all elements in a data structure without exposing the internal representation. Recall that an **iterator** is an object that encapsulates a reference to an object in the internal representation of a data structure. An iterator is provided to an application program to enable it to traverse the data structure, and perhaps remove selected elements during the iteration.

For a given iterator for a collection, we say that a **concurrent modification** has occurred if a change is made to that collection since the time that the iterator was created, where the change is not a direct result of a method call on that iterator. Java's iterators handle concurrent modification by a **fail-fast** mechanism. Namely, if an iterator experiences a concurrent modification, any subsequent method invocation on that iterator will throw a *ConcurrentModificationException*. When a data structure is modified through any means other than the iterator itself, the resulting exception can be difficult to debug because while exceptions usually help pinpoint the cause of the failure, here the exception occurs in the "wrong" place. A loop might be advancing through the list, and then, through no fault of its own, get a concurrent modification exception because somewhere else in the program the list was modified. Finding out where that modification occurred could be difficult. One might prefer that the modification itself be prevented while the iteration was in progress, but that would require keeping track of all iterators and knowing when they were "finished." Furthermore, a correct method might use an iterator to traverse only part way through a data structure, until finding a particular value, for example. One would not want the execution of such a method to prevent all further modifications to the data structure.

One possible cause of a concurrent modification exception is true concurrency, in which multiple threads may be using the same data structure simultaneously. Such problems could be avoided by synchronizing the threads using locks, as discussed in the previous section. However, a concurrent modification exception can occur even within programs that contain only one thread. For example, suppose we want to traverse a list of people, removing all people from the list who are incompatible with another person in the list.

```
Iterator it1 = personList.iterator();
while (it1.hasNext()) {
    Person p = (Person) it1.next();
    Iterator it2 = personList.iterator();
    while (it2.hasNext()) {
        Person q = (Person) it2.next();
        if (p.dislikes(q))
            it2.remove();
    }
}
```

If, at any point during the execution of this loop, a person who dislikes another person is discovered, the *remove* method would be called on the second iterator, modifying the data structure. At the next iteration of the outer loop, the first iterator would throw a *ConcurrentModificationException*. The same problem arises if an iterator is retained in a variable to keep track of a particular location in a list. Iteration cannot be resumed through that iterator if there are intervening modifications of the data structure.

The iterators presented in this book minimize the problem of concurrent modification by implementing the data structures so that the iterators can continue to operate whenever possible, even if the data structure has been modified. Since certain mutations, like swapping the positions of two elements, would certainly disrupt the iteration order, we make a distinction between mutators that

do not cause such problems. We refer to any mutator that would prevent iteration from completing properly as a **critical mutator**. It is important that *hasNext* and *next* are called atomically with respect to any mutations of the collection, because otherwise a *NoSuchElementException* could occur during iteration. For example, if an iterator is positioned just before the last element of a collection, *hasNext()* would return *true*. However, if the last element is subsequently removed, calling *next()* on the iterator would result in a *NoSuchElementException*. Iteration order and concurrent modifications are discussed in much more depth in Section 5.8.5. But first we introduce the concept of a locator, whose implementations include markers (that refer to a position) and trackers (that keep track of the position of an element).

5.8 Locator Interface

Iterator<*E*>, *Cloneable*

↳ *Locator*<*E*>

Many applications need to remember a location within a collection. For example, an application program may need to advance through a collection to perform some computation on each element. Another application may need to insert an element into a collection and later go directly to that element in the collection (in order to remove it, for example) without incurring the cost of searching the collection for that element. As discussed in Section 5.7, one could imagine meeting these requirements by providing the application with references to objects in the internal representation of the data structure, but doing so would expose the internal representation and violate encapsulation. Consequently, it is important to provide the user with a mechanism for remembering a location within a collection without exposing the internal representation.

5.8.1 Case Study: Maintaining Request Quorums for Byzantine Agreement

When critical applications require robustness against failures, such as crashes and network outages, it is necessary to replicate processing on multiple computer hosts that continue to provide service as a whole even if some limited number of the replicas fail. The most difficult failures to tolerate are the so-called **Byzantine failures** [99] in which one or more hosts may be taken over by an attacker and begin to send erroneous or conflicting messages in order to disrupt correct service. One algorithm for handling such failures [38] involves a *primary* server replica that receives client requests and then forwards them to other replicas in the server group for processing. (Since the primary itself may be faulty, the algorithm includes a protocol for electing a new primary.) When the client that sends these requests is also replicated for fault tolerance [120, 151], it is necessary for the server primary to collect a **quorum** of sufficiently many matching signed requests before forwarding the request to the other server replicas for processing. The quorum is necessary to prevent a faulty client replica from forcing the server to execute a request that has not been agreed upon by the other client replicas. This scheme implies that the server primary must collect requests from the client replicas until a quorum has been received. However, a faulty client replica may seek to fill up the primary's memory by flooding it with requests not agreed upon by the others. Consequently, we wish to limit the number of outstanding requests the primary is willing to entertain from each client replica.

The problem that must be addressed is to find a way for the server primary to collect signed requests from client replicas and detect when a quorum of *q* matching requests has been received. Furthermore, the server primary must retain only the *k* requests most recently received from each client replica. That is, when request *m + k* is received, request *m* should be discarded even if a quorum has not yet been obtained for that request. To simplify processing, each request has a

calculated **digest** that is assumed to be unique to that request and consistent across all (nonfaulty) client replicas.

In selecting the ADTs to use for this problem, there are two important considerations. First, for each digest, we must collect corresponding requests until there are at least q such requests. Second, we need an efficient way to discard the oldest requests sent by each client replica to prevent memory overruns due to flooding attacks. We begin with the request collection problem, and then move to the discard problem. The fact that each request is identified by a unique digest suggests the use of a tagged bucket collection, where the tag is the digest and the associated value is the collection of requests with that digest received so far. This way, when the size of the collection for a particular digest reaches q , it is easily detected that a quorum has been reached and that the collection can be forwarded to the other server replicas. To avoid counting duplicate requests from the same client replica twice, we want the collection associated with each tag to contain only one request per client replica. Therefore, we want this collection to be a set. Neither the tags nor the requests in the sets need to be ordered among themselves.

For the discard problem, we want to efficiently remove a client replica's oldest request from the appropriate request sets when a new request arrives. More specifically, if the primary receives a request from a given client replica, it must identify (1) whether the primary is currently holding k requests from that replica, and if so (2) remove the oldest such request. Searching through all of the request sets would be prohibitively expensive, and would require extra information on each request to determine its age. It is necessary to maintain some historical information about each client replica. To do so, we could keep for each client replica a sequence of up to k digests, in order of age. Then, when the sequence is full, the oldest digest could be used to retrieve the corresponding set of requests, and then we could search through that set to remove the client's request. However, it is possible to avoid this search by instead keeping up to k trackers, in order of age, that refer to elements in the request sets. This way, when a new request arrives, we can simply call the remove method on the oldest tracker.

To summarize, we require a tagged collection that maps digests to sets of requests. In addition, we require a tagged collection that maps each client replica identifier to a bounded queue of trackers that refer to its most recent k requests. In selecting the data structure, since ordering is not necessary, we use a bucket mapping from digests to sets of requests. The mapping from each digest to a set of requests should use an equivalence tester based on the sender replica id. Finally, this application will use a mapping from the sender's replica id to a bounded queue of size k containing references to requests.

5.8.2 Interface

To specifically support iteration over a data structure without exposing the internal representation, we build on Java's *Iterator* interface, which has the following methods:

boolean hasNext(): Returns *true* when the iteration has more elements.

E next(): Returns the next element in the iteration, and advances the iterator. Calling this method repeatedly until *hasNext()* returns *false* will return each element in the collection exactly once.

void remove(): Removes the last element returned by the iterator, leaving the iterator just before the item removed. A subsequent call to *next* would return the item just after the one removed.

Java collections typically provide an *iterator* method that returns an iterator that is logically positioned just before the first element of the collection. The method *next* provides access to the elements, but also has the side effect of advancing the iterator. To support additional flexibility for

applications, we define a *Locator* interface that extends *Iterator* with methods that separate navigation from element access. Also, methods are included to provide the application program with flexibility in controlling when concurrent modification exceptions occur, as described further in Section 5.8.5. Locators provide a method to get the element (currently) associated with the locator, independent of methods that move the locator forward or backward through the collection. The semantics of these additional methods are as follows. **FORE** refers to the position before the first element, and **AFT** refers to the position just after the last element.

boolean advance(): Advances to the next element in the collection (if there is one) and returns *true*. If the locator is already at the last element of the collection then *false* is returned and the locator is moved to AFT. If a call to *advance* is made when the locator is at AFT, an *AtBoundaryException* is thrown. Starting with the locator at FORE and calling *advance* repeatedly until *false* is returned will reach each element in the collection exactly once.

E get(): Returns the element associated with this locator. When a tracker is used, the element associated with the locator might no longer be in the collection. If desired, the *inCollection* method can be used to determine if a tracked element is currently in the collection. If the locator is at FORE or AFT then a *NoSuchElementException* is thrown.

void ignoreConcurrentModifications(boolean ignore): If *ignore* is *true* then concurrent modification exceptions are disabled. If *ignore* is *false*, it sets the state of the iterator so that (future) concurrent modifications will be noticed and result in concurrent modification exceptions. The semantics of this method is described in more depth in Section 5.8.5.

void ignorePriorConcurrentModifications(): Any prior concurrent modifications are ignored, but if another critical mutator is executed then any later access will result in a concurrent modification exception. This is different from *ignoreConcurrentModifications*, which will continue to tolerate critical mutators without throwing any concurrent modification exceptions. See [Section 5.8.5](#) for further discussion.

boolean inCollection(): Returns *true* if and only if the locator is at an element of the collection.

boolean retreat(): Retreats to the previous element in the collection (if there is one) and returns *true*. If the locator is already at the first element of the collection then *false* is returned and the locator is moved to FORE. If a call to *retreat* is made when the Locator is at FORE, an *AtBoundaryException* is thrown. Starting with the locator at AFT and calling *retreat* repeatedly until *false* is returned will reach each element in the underlying collection exactly once, in the reverse order.

5.8.3 Markers and Trackers

There are two different approaches we use to implement the *Locator* interface, a marker and a tracker. A **marker** marks a location within the internal representation of the collection, and continues to mark that location even if a different element is placed there. In contrast, a **tracker** keeps track of the location of a particular element in the collection, even if the element is moved to another location. We call this **tracking** the element.

To understand the difference between a marker and a tracker, consider a valet parking service. Suppose that when the attendant parks a customer's car (i.e., an insertion into the collection), the customer receives a receipt consisting of a sealed envelope containing the number of the parking space. When the customer is ready to leave, the attendant looks inside the sealed envelope to efficiently locate the car. This kind of receipt is analogous to a marker because it identifies a particular location within the data structure, in this case a parking space number. (Although the customer

holds the envelope, its contents are available only to the attendant, so the internal representation of the data structure is not exposed.)

Now, suppose that the parking lot is so small that some cars are parked to block others. In this case, the attendant must sometimes move cars from their original parking spaces in order to retrieve blocked cars. When rearrangement of cars is a possibility, providing the customer with a sealed envelope recording the original location of the car is inadequate for retrieving it later. We require something that maintains the current location of the car, even when it is moved as a side effect of another operation. So, instead of giving the customer a sealed envelope, the attendant could give the customer a receipt in the form of an electronic device that initially holds within it the number of the original parking space. Then, whenever the customer's car is subsequently moved, the contents of the electronic device is automatically updated. When the customer is ready to leave, the attendant accesses the electronic device to efficiently locate the car. This kind of receipt is analogous to a tracker because it identifies the location of a particular element within the data structure. Again, although the customer holds the device, its contents are available only to the attendant, so the internal representation is not exposed. Note that if the customer loses the device, the car can still be located, but only by a more time consuming search through the parking lot.

Marker semantics: The semantics for a marker are consistent with those for a Java iterator. When *advance* is called on a marker, it moves to the next location, and when *retreat* is called on a marker it moves to the previous location. So *next* is equivalent to calling *advance* followed by *get*. When *next* returns the last element, then logically the iterator is at AFT. The advantage of a marker over an iterator is that it separates the movement of the marker from retrieving or modifying the element at the current location.

As described for the *Iterator* interface, when the *remove* method is called on a locator, it is positioned just before the item that was removed. Because a marker refers to a location, its location after the removal is the location of the previous element (the one before the removed element in iteration order), so a subsequent call to *get* would return that previous item.

Tracker semantics: A tracker is designed to track a particular object until *advance* or *retreat* is called. When we say that a tracker **tracks** an element, we mean that it tracks an occurrence of the given element. In particular, if that occurrence of the element is removed, *inCollection* will return *false* even though there may be another occurrence of the element elsewhere in the collection. When *advance* is called on a tracker it begins tracking the next element, and when *retreat* is called on a tracker it begins tracking the previous element. When *remove* is called on a tracker, the tracker is considered to remain *between* the removed element and the previous one. Thus a subsequent call to *get* results in a *NoSuchElementException* since the element being tracked is no longer in the collection. For all locators (both markers and trackers), calling *advance* after a removal positions the locator at the element after the item removed.

Overhead: Trackers and markers both incur some overhead. First, there is the space overhead for the locator itself. A marker is only updated only as a direct result of method calls on the marker itself, so the overhead incurred for the other methods is the same as for an ordinary Java iterator. For pointer-based data structures a tracker is simply implemented as a reference to a node of the data structure, so as with the marker there is no overhead for the other methods. For some array-based data structures, there is a negligible amount of overhead associated with maintaining the trackers, and for others the cost can be significant.

In Section 7.2, we define the *Tracked* interface. This interface includes the *addTracked* method, which returns a tracker to the newly inserted element. When no tracker is needed, the *add* method can instead be used to avoid the overhead of creating and maintaining the tracker. We do not implement the *Tracked* interface for any data structure for which the support of a tracker would incur more than a small additive constant cost in the time complexity. For all data structures we present, a locator can also be obtained using the *iterator()*, *iteratorAtEnd()*, or *getLocator(E element)* method.

Benefits of trackers: One may wonder whether trackers are worth the additional overhead and complication. However, when finding the location of an element in a collection is a frequent op-

eration, trackers can improve application performance dramatically. The most common use for a tracker is to enable constant-time access to a specified element, where the application program retains the tracker returned when that element was inserted into the collection. As a simple example, suppose that a tracked doubly linked list is being used to manage a list of jobs that are waiting for a resource. Suppose that job J on the list no longer needs the resource, and should therefore be removed from the list. One could search for job J in the list, but the time to find job J would be proportional to its position in the list. However, suppose the application program retains the tracker returned when each job is inserted into the list. For example, the application could keep a mapping from jobs to trackers. Then, when J no longer needs to wait for the resource, the mapping could be used to retrieve the tracker for J in constant time, which could then be used to locate and remove J from the list, again in constant time.

A tracker can also be used when an application wants to combine two data structures in order to obtain the benefits of each. For example, suppose an application wants to maintain a set of jobs in a way that supports constant-time access to either the highest priority job or the job that has been waiting the longest. Such an application can be best supported by combining a Queue ADT and a PriorityQueue ADT. A very simple implementation is to independently maintain a queue and a priority queue, where each element maintains a tracker to its location in both data structures. Then if the element with the highest priority is removed, in constant time it can be located and then removed from the queue. Likewise, if the element at the front of the queue is removed, in constant time it can be located in the priority queue. Since most PriorityQueue implementations take linear time to locate an element and only logarithmic time to remove an element once it is located, the ability to locate the desired element in constant time is important.

Trackers can also lead to significant improvements in the asymptotic time complexities of common algorithms. One example is Dijkstra's algorithm to find the shortest path in a weighted graph (see Section 57.2). Another example is Prim's algorithm to compute a minimum spanning tree of a weighted graph (Section 57.3). In both of these algorithms, a priority queue is used that holds the vertices in the graph. Whenever vertex v is added to the solution, it is removed from the priority queue. Also the priority for vertices that have an edge from v may change. While there are very efficient implementations for a priority queue, it typically takes linear time to locate an element within the data structure. Thus, in order for Prim's algorithm and Dijkstra's algorithm to be as efficient as possible, it is crucial that in constant time a vertex can be located in the priority queue. This can be achieved by using a tracked implementation for the priority queue, and storing a tracker for each vertex. Since the implementation of Prim and Dijkstra's algorithms already maintain information about each vertex, it is easy to add an instance variable in each vertex that holds the tracker returned by the *addTracked* method.

5.8.4 Iteration Using Locators

This section discusses one of the most common uses for a locator, namely to iterate through a collection. Iteration can be carried out using the *Iterator* methods, as illustrated in Section 5.7. Alternatively, the following example uses the additional *Locator* methods to illustrate removing all the “bad” elements from a collection.

```
Locator loc = collection.iterator();
while (loc.advance()) {
    System.out.println("Processing " + loc.get());
    if (badElement(loc.get()))
        loc.remove();
}
```

Because the *Locator* methods separate element access from navigation, multiple calls to *get* can occur within one iteration of the loop without any side effects on the position of the locator. This is in contrast to *next*, which should be called only once in each iteration of a loop.

When iterating through a collection, each iteration of the loop will visit exactly one element (through its calls to *get* or *next*) and no two iterations will visit the same element*. Of course, if the loop terminates early, not all items will be visited. We say that the order in which the elements are visited in the forward direction is the **iteration order**.

It is also possible to use a locator to traverse a collection in the reverse direction, starting from the end.

```
Locator loc = collection.iteratorAtEnd();
while (loc.retreat()) {
    System.out.println("Processing " + loc.get());
    if (badElement(loc.get())) {
        loc.remove();
        loc.advance();
    }
}
```

Recall that markers position themselves at the previous item after removing an element. Therefore, as shown in the above example, it is necessary to advance the marker after a remove when iterating in the reverse direction. Otherwise, the element just before the removed item could be skipped. Advancing is not technically necessary if the locator is a tracker, since the tracker is between items after a removal. However, it is good practice to do so anyway, especially if the declared type of the variable is a *Locator*, since a marker could be substituted for the tracker in a later version of the implementation.

Recall that a marker just marks a location within the data structure. So unless it is at FORE or AFT, *inCollection()* returns true. On the other hand, a tracker can be between two positions (for example, because the element it was tracking was been removed). When called on a tracker, *inCollection()* returns *true* unless the tracker is at FORE or AFT, or is tracking an element that has been removed. The *get* method will throw a *NoSuchElementException* exactly when *inCollection* returns *false*.

5.8.5 Iteration Order and Concurrent Modifications

We have seen that a locator can be used to iterate over a collection, and to modify the collection during iteration. However, we have not discussed the semantics of continuing iteration even if changes are made to the data structure through other means.

Concurrent modifications can affect the order in which elements are visited during iteration. For example, consider a tracker that tracks an element currently at position 5 in a positional collection. If, through some method call on the collection, the element is swapped with the element at position 9, continued iteration with that tracker would skip over the elements at positions 6 through 8. A similar problem could occur with a marker at position 5 if, for example, the first three elements of the collection were removed.

To detect concurrent modifications, each data structure keeps a modification counter that is incremented on each mutation of the data structure. Each iterator stores a copy of that counter when it is

*To simplify our discussion, we consider each element in the collection to have its own identity, even though two elements may actually be references the same object. In other words, as we discuss iterating over the elements of a collection, we consider each occurrence of an item in a collection to be a different element.

created, and updates its own counter to match that of the data structure with each mutation that is made by the iterator (e.g., a *remove*). However, if the iterator ever observes that the data structure's count is higher than its own, it throws a *ConcurrentModificationException*.

For some applications it is desirable for locators to continue to work even if there have been concurrent modifications. For example, consider a (properly synchronized) shared queue in which items may be added at the back and removed from the front, and suppose that the queue is large and changes are made with high frequency. Now, suppose that one thread wishes to iterate over the contents of the queue, for the purpose of displaying it to an administrator. One possibility for avoiding concurrent modifications would be to place a synchronized block around the iteration loop, but the iteration may take a long time relative to the other operations, and other threads would be blocked from adding and removing from the queue during that time. Another possible implementation might catch the *ConcurrentModificationException* and start the iteration over again, but since the update frequency is high, the iteration may never complete. One might also imagine making a "snapshot" copy of the data structure and iterating over that copy, but it would take time to make the copy, during which other threads may be blocked, and the copy would consume additional memory. Finally, invalidating locators for every concurrent modification would essentially rule out the use of a tracker, since it is designed to provide constant-time access to the tracked element, even the data structure is modified.

When using a locator to iterate forward or backward through a collection, we say that the iteration order is **consistent** if and only if

1. the locator visits each of the persistent elements exactly once, where the **persistent elements** are those that were in the collection when the locator was created and remained in the collection throughout the use of the locator, and
2. if the ADT specifies a particular iteration order (such as a *PositionalCollection* or *OrderedCollection*), that order will be respected for all the elements that are visited.

Note that elements that were added or removed during iteration may or may not be included in the iteration order. For example, a locator iterating over a positional collection would visit newly added items only if the locator had not yet reached their positions at the time they were added.

Certain concurrent modifications, such as a *swap* in a positional collection (see Chapter 9) or resizing a hash table (see Chapter 23), may rearrange the data structure in a way that would prevent a locator from being able to guarantee consistent iteration order. A mutator is said to **invalidate** a locator if it prevents the locator from guaranteeing a consistent iteration order. Similarly, we say that a locator is **stale** when it has been invalidated by a mutator. To distinguish mutators that invalidate locators from those that do not, we refer to the mutators that could invalidate a locator as **critical mutators**, and provide a list of these for each ADT.

In contrast to Java's fail-fast iterators, the locators we present are resilient to certain concurrent modifications. Trackers continue to refer to their elements, even if concurrent modifications are made. Moreover, iteration (through *advance* and *retreat*) will result in a consistent iteration order. A *ConcurrentModificationException* will be thrown only if, as the result of the execution of a critical mutator, consistent iteration order cannot be guaranteed. A locator guarantees that iteration will continue with a consistent iteration order, and without a concurrent modification exception, provided that no modifications are made through critical mutators. Note that although a certain method of an ADT may be designated a critical mutator, each implementation of that ADT is free to guarantee that consistent iteration order is maintained even after that mutator executes. Therefore, a locator may not necessarily throw a concurrent modification exception even after a designated critical mutator is called. For example, see the *add* method for the separate chaining implementation of a *Set* (Chapter 23).

In certain circumstances, an application may wish to continue using a locator to iterate through a collection even though the iteration order is not necessarily consistent. For example, an applica-

tion may want to iterate forward from a tracker that was saved long ago. The fact that concurrent modifications have occurred through critical mutators since the time the tracker was created is of no consequence to the application. It may simply need to perform an operation on the tracked element, or visit the item that is currently immediately before or after the tracked element. To suppress concurrent modification exceptions and allow iteration to continue even if the order may not be consistent, our locators provide two methods: *ignoreConcurrentModifications* and *ignorePriorConcurrentModifications*. The first allows the application to suppress notification of concurrent modifications, possibly temporarily, so that iteration can continue even if the iteration order may not be consistent (elements may be visited twice, skipped, or seen out of order). The second resets the locator to ignore prior concurrent modification so that the iteration order from this point on will be consistent, and calls to critical mutators in the future may result in a *ConcurrentModificationException* from this locator. Both of these can be found in the *AbstractLocator* inner class of the *AbstractCollection* class (Chapter 8).

5.9 Version Class

package collection

Each data structure will have one instance of the *Version* class. Each *Version* instance has a single instance variable to maintain a modification count that tallies only those modifications that would invalidate a locator. Each locator holds its own version number, an integer that is initialized with the latest modification count. The locator implementation has the option of updating its own copy of the version number when the locator calls a mutating operation on the data structure. In each public method of the locator, the *check* method is used to compare its own version number with that held in the *Version* object of the data structure. If the comparison reveals that the locator is stale, a concurrent modification exception is thrown. Unlike Java's fail-fast iterators, the locator implementations presented in this book are aggressive about preserving the validity of locators to avoid concurrent modification exceptions. Therefore, the version is incremented only when a critical mutator is called. The version number is not incremented after executing any non-critical mutator.

```

int modificationCount = 0;

public int increment() {
    return ++modificationCount;
}

public int getCount() {
    return modificationCount;
}

public void restoreCount(int count) {
    modificationCount = count;
}

public void check(int v) {
    if (v < modificationCount)
        throw new ConcurrentModificationException("stale locator");
}

```

5.10 Visitors for Traversing Data Structures

While iterators provide the ability to traverse a collection without exposing its internal representation, they have several disadvantages. First, each iteration typically involves the creation of a new object. If a program performs a lot of iteration, this can increase the amount of time spent by the garbage collector. Second, iterators can be difficult to write, since their state must capture enough information to continue the iteration from its current point in the iteration order. This is especially problematic for data structures in which the desired iteration order is most easily expressed using a recursive algorithm (such as an inorder traversal of a tree or depth-first traversal of a graph). In such cases, the state of the iterator might need to include a stack. Third, since iterators provide element-by-element access to the collection, there is no way for the implementation to ensure that the entire collection is traversed in a single atomic step. The documentation could stipulate that users should lock the entire collection during iteration, as described in Section 5.6, and the implementation could throw concurrent modification exceptions when violations occur. However, because iterators are returned from the collection, it is not possible for the implementation of the collection to directly enforce that a lock be held throughout iteration.

All three of the above disadvantages result from the fact that iterators support external iteration over the internal structure of the collection. However, we can avoid these disadvantages by turning the design “inside out.” Rather than provide an iterator for use by external code, the external code can implement the following *Visitor* interface.

```
public interface Visitor<T> {  
    public void visit(T item) throws Exception;  
}
```

Each collection can provide an *accept* method that takes a visitor as a parameter. The obligation of the *accept* method is to call the visitor’s *visit* method once for each element in the collection, in accordance with the iteration order. Any existing object could implement the *Visitor* interface, avoiding the need to create a new object for each traversal of the collection. Furthermore, a standard loop or recursive implementation of the *accept* method is sufficient, since the state of the iteration need not be captured for later continuation. Finally, the *accept* method could be synchronized so that a lock is held on the collection for the duration of the traversal.

One might imagine entirely doing away with iterators in favor of visitors. However, iterators do have certain advantages:

1. They do not require that the user implement a special interface in order to traverse a collection.
2. They allow the user to gracefully stop the iteration at any point and resume later.
3. Certain types of iterators (see [Section 5.8](#)) can provide a mechanism for keeping track of the location of objects within the collection, which is critically important for the performance of certain algorithms such as Dijkstra’s Shortest Path Algorithm (Section 57.2) and Prim’s Minimum Spanning Tree Algorithm (Section 57.3).

Therefore, the collection implementations in this book support both iterators and visitors.

Chapter 6

Partition ADT and the Union-Find Data Structure

package partition

PartitionElement<*T*>

A partition is a division of a group of objects into disjoint sets that we call **components**. For example, voters in a political primary election may be partitioned into political parties, or kosher foods may be partitioned into groups that contain meat, dairy, or neither. One could also create a partition of the vertices of a graph into connected components, such as cities that are connected with each other by waterways. The criteria for creating partitions are generally symmetric and transitive. For example, it would *not* make sense to create a partition of cities into components that consist of cities within 500 miles of each other, or that are reachable from each other city by a direct airline flight.

We view a partition abstractly as a collection of disjoint components whose union is the total collection. However, the actual data type we use to represent the partition is not the partition as a whole but rather the individual components of the collection. In fact, we do not even have a special entity that corresponds to each component. Instead, each component in the partition is defined by an (arbitrary) element of that component known as the **representative element**, or simply the **representative**. So the data type that we implement for the Partition ADT is simply that of a *PartitionElement* that represents one element of the partition.

This book presents two very different applications for the Partition ADT:

- Section 6.8 illustrates how the Partition ADT can be used to support merging two data structures without invalidating their trackers. The overhead introduced is nearly constant, so for data structures in which a constant time merge is possible, the resulting merge that preserves trackers also runs in constant time.
- Section 57.4 uses a partition to maintain a set of connected components in a graph for Kruskal's minimum spanning tree algorithm.

6.1 Partition Element Interface

The *PartitionElement* interface includes the following methods.

***PartitionElement*(*T applicationData*):** Creates a new partition element that has the given application data as its associated data.

***PartitionElement*<*T*>*.findRepresentative*():** Returns the representative element for the component that includes this partition element.

***T get*():** Returns the data associated with this partition element.

boolean sameComponent(PartitionElement<T> x): Returns *true* if and only if this partition element and *x* are in the same component.

void set(T applicationData): Sets the associated data for this partition element to the provided value.

PartitionElement<T> union(PartitionElement<T> x): Combines the components holding this partition element and *x* into a single component. It returns the representative element for the resulting component. If this element and *x* are already in the same component, no mutation occurs and their representative is returned.

6.2 Selecting a Data Structure

The selection process is straightforward for this ADT. The union-find data structure (Section 6.3) supports all of the required methods in constant amortized time with efficient space usage, so it is not necessary to consider alternative implementations. Technically, the time complexity grows very slowly with the number of components, but for all practical purposes this multiplicative constant is at most 4. See [Section 6.7](#) for more details.

6.3 Union-Find Data Structure

UnionFindNode<E> implements *PartitionElement<E>*

Uses: Java references

Used By: LeftistHeap (Chapter 26), Kruskal's minimum spanning tree algorithm (Section 57.4) of AbstractWeightedGraph (Chapter 57)

Strengths: Very space and time efficient. For applications where the limited functionality is sufficient, union-find is an ideal data structure.

Weaknesses: Very limited functionality is supported. In particular, it is not possible to iterate over the partition, iterate over all elements in a given component, search for an item, or remove an element from a component.

Competing Data Structures: When it is necessary to iterate over the elements of a component, consider combining this data structure with another basic data structure to provide efficient access to the set of elements that are in the same component of the partition. For example, the union-find data structure could be combined with the singly linked list (Chapter 15) data structure. In particular, a list could be maintained for each component to support iteration within the component. Each representative element would keep a reference to the list for its component. When a union occurs, the lists of the two representatives would be merged. Interestingly, the union-find data structure can be used within the `SinglyLinkedList` class to support merging two singly linked lists, as discussed in more depth in the Section 6.8. This combination of union-find and linked lists would allow iteration and therefore linear time search within a component.

6.4 Internal Representation

One might imagine a simple approach to maintaining a partition in which a list of elements is maintained for each component. In that solution, each partition element could have a reference to its list or to the representative element for its group. The space usage for such a solution would require three references per element (one for the next element in the component, one for the associated application data, and one for the representative element). This implementation would allow constant time support for *findRepresentative* since a single reference provides this value. Similarly, *sameComponent* method could be implemented in constant time by comparing the representative element for the two components. However, the cost for *union* would be linear in the size of the smaller component. While the lists can be combined in constant time by having the last element in one list reference the first element of the other list, all of the representative elements in the smaller component would need to be updated.

The union-find data structure provides a better alternative solution in which each element requires only two references and for which all methods, for practical purposes, run in amortized constant time. The only advantage of the list-based approach described above is that it would be possible to iterate over all of the elements in the same component as a specified element.

The union-find data structure represents the partition as a **forest of in-trees** where each tree in the forest directly corresponds to a component of the partition. An advantage of this representation is that the tree structure is kept in a distributed fashion with each partition element only maintaining its parent. There is no centralized data structure with references to all of the elements. The distributed representation supports the required methods very efficiently. This representation does not provide any means to find a child or sibling of an element, but such capabilities are not needed for the required operations of the ADT.

Terminology: Throughout this section we use the following terminology.

- For partition element x , we define $T(x)$ to be the tree that contains x . A partition is composed of disjoint sets in which each element is in exactly one component, so for each x , $T(x)$ always exists and is unique.
- We define the **representative element** for partition element x to be the root of $T(x)$.
- We define the **height** of a tree T , denoted $height(T)$, to be the maximum over all nodes $x \in T$ of the number of *parent* references that must be followed to reach the root of T from x .

Instance Variables and Constants: There are three instance variables associated with each union find element. First, *applicationData* is a reference to the object associated with the partition element. Second, *parent* is a reference to the partition element that is the parent. We use the convention that the root of a tree is its own parent. Finally, the *rank* of a node x , which is only used in the algorithm when x is a root, provides an upper bound on the height of $T(x)$.

```
T applicationData;
UnionFindNode<T> parent;
int rank;                                     //upper bound on tree height
```

Populated Example: Figure 6.1 shows a populated example of the union-find data structure for the partition $\{\{B, C, D, G\}, \{A, E, F\}\}$.

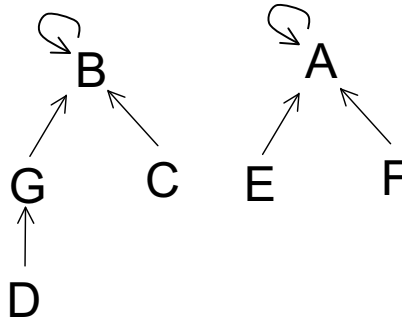


Figure 6.1

A populated example of the union-find data structure for the partition $\{\{B, C, D, G\}, \{A, E, F\}\}$. The parent reference is shown using an arrow directed upward towards the parent. The references to the application data and the ranks are not shown. The rank of B is 2, and the rank of A is 1.

Abstraction Function: Let $\{T_1, \dots, T_k\}$ be the trees in the union-find representation. Then the partition consists of k components where component i contains exactly the elements $x \in T_i$.

Design Notes: The union find node illustrates the **proxy design pattern** by having the representative element of each component serve as a surrogate for the entire component. More specifically, in the application for union-find described in Section 6.8, the representative union find node for each component refers to the data structures that resulted from combining the data structures associated with all of the other elements in that component.

There is no provision for explicitly removing elements. Because the representation is not centralized, a given partition element remains in the structure only if the application holds a direct reference to it, or if it is reachable by an in-tree path from a partition element to which the application refers. Otherwise, the partition element is automatically eligible for garbage collection.

Optimizations: If efficient iteration is desired without the overhead of a singly linked list, then a *next* field could be added to each union-find node which would reference the next element in that component. In addition, a Locator inner class would need to be provided to support the iteration. We refer to such a solution in which a linked list for iteration is integrated into the data structure as a **threaded data structure**.

6.5 Representation Properties

We introduce the following properties where $\{T_1, \dots, T_k\}$ are the components, and X is the set of all elements in the partition.

ROOT: For each element $x \in X$, if x is the root of $T(x)$ then $x.parent = x$.

ROOTRANK: For each root r , $r.rank$ is an upper bound for the height of $T(r)$. More formally, for each element $x \in X$, if $x.parent = x$ then $height(T(x)) \leq x.rank$.

PARENTRANK: For each node that is not a root, its rank is strictly less than that of its parent. More formally, for each element $x \in X$, if $x.parent \neq x$ then $x.rank < x.parent.rank$.

PARTITION: Each element $x \in X$ is in exactly one of $\{T_1, \dots, T_k\}$. More formally, we require that $T_1 \cup T_2 \dots \cup T_k = X$ and for any T_i and T_j where $i \neq j$, $T_i \cap T_j = \emptyset$.

6.6 Methods

The constructor takes *data*, the application data, as its parameter. It creates a new component holding a single union-find node associated with the given application data.

```
public UnionFindNode(T data) {
    this.applicationData = data;
    rank = 0;
    parent = this;
}
```

Correctness Highlights: Initially each element is a singleton node that is a root. Since, by definition a singleton node has height 0, ROOTRANK is satisfied by setting *rank* to 0. ROOT is satisfied by setting the parent reference to itself, since the new element is the root of its tree. PARENTRANK vacuously holds since the new node is a root. Finally, PARTITION is preserved since the new element is placed in exactly one tree. Thus it is in exactly one component. ■

The *toString* method returns a String that shows the application data held within the node.

```
public String toString() {
    return "" + applicationData;
}
```

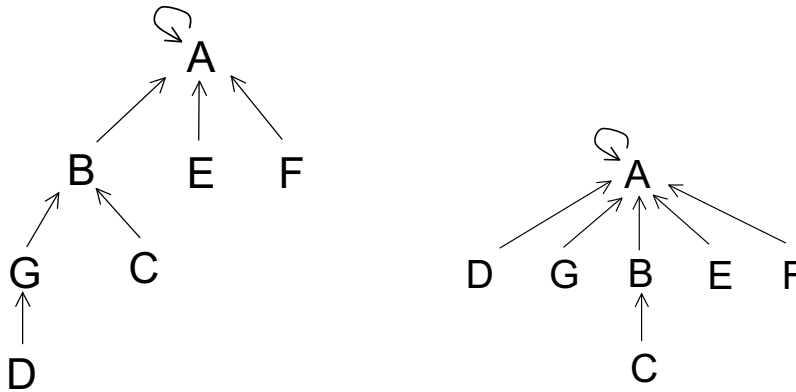
The *get* method returns the application data held within the node.

```
public T get() {
    return applicationData;
}
```

The *set* method takes *data*, the new value for the application data. It resets the application data to the given value.

```
public void set(T data) {
    this.applicationData = data;
}
```

The *findRepresentative* method returns the representative node for the component containing this node. The purpose of the tree structure is solely to find the representative element for any given node. When a path of *parent* references is traversed to find the representative element, all of these

**Figure 6.2**

The tree shown on the left is modified, to produce the tree on the right, by path compression that occurs when *findRepresentative* is called for node *D*.

parent references are directly set to the root. This process, called **path compression**, improves the overall efficiency of this data structure from logarithmic to nearly constant amortized cost. Figure 6.2 illustrates the result of path compression.

```
public UnionFindNode<T> findRepresentative() {
    if (this != parent)
        parent = parent.findRepresentative();           //perform path compression
    return parent;
}
```

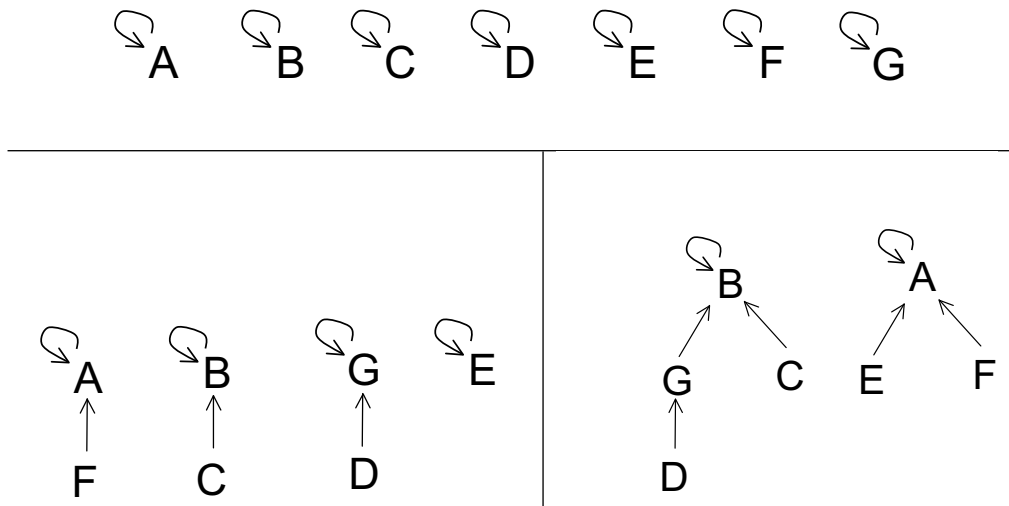
Correctness Highlights: By ROOT, the recursion will end when the method is called on the root. ROOT is not affected by this method since the parent of the root is not changed. The representative element does not change for any element, so PARTITION is preserved.

We now argue that ROOTRANK and PARENT RANK are preserved. Let x be the element on which this method is called. Observe that the height of $T(x)$ either remains the same or is reduced by this method. Thus for the root r of $T(x)$, ROOTRANK is preserved. Let $x \rightarrow x_1 \rightarrow \dots \rightarrow x_p \rightarrow r$ be the path on which the path compression is performed. By PARENT RANK, $x.rank < x_1.rank < \dots < x_p.rank < r.rank$. Thus by transitivity, PARENT RANK is preserved when each of $x, x_1, \dots, x_p$ are updated so that r is their parent. ■

The *sameComponent* method takes x , a union-find node, as its parameter. This method returns *true* if and only if x and the node on which this method is called are in the same component.

```
public boolean sameComponent(PartitionElement<T> x) {
    return x != null && x.findRepresentative() == findRepresentative();
}
```

The *union* method's parameter is x , a node whose component is to be combined with the component of this partition element. It combines the components of x and this partition element into one component. The method returns the representative element of the resulting component. This method throws an *IllegalArgumentException* when x is not a union-find node. If x and the node on which this method is called are in the same component then no change is made, and the root of their component is returned.

**Figure 6.3**

Top: Union-find data structure initialized with union-find nodes A-G. *Bottom Left:* The same structure after $C.union(F)$, $B.union(C)$, and $G.union(D)$. *Bottom Right:* Finally, after $C.union(G)$ is executed.

The *union* method first finds the representative node for the component holding x and for the component holding the node on which this method is called. If these two nodes are already in the same component then the common root is returned. Otherwise, one root becomes the child of the other. To obtain the best amortized time complexity, **union by rank** is applied in which the root with the larger rank becomes the root of the resulting component, with ties broken arbitrarily.

Figure 6.3 illustrates the results of executing a sequence of *union* operations, beginning with a forest initialized to have the 7 nodes A, B, C, D, E, F, and G. After $A.union(F)$, $B.union(C)$, and $G.union(D)$ are executed, the structure has four components. Then, after $E.union(F)$ and $C.union(G)$ are executed, there are only two components.

```

public PartitionElement<T> union(PartitionElement<T> x) {
    if (!(x instanceof UnionFindNode))
        throw new IllegalArgumentException("union-find node expected");
    UnionFindNode<T> r1 = (UnionFindNode<T>) x.findRepresentative();
    UnionFindNode<T> r2 = this.findRepresentative();
    if (r1 == r2)
        return r1; //already in some component, no change needed
    else { //only merge if x is in a different component
        if (r1.rank > r2.rank) { //link the roots to preserve Rank
            r2.parent = r1;
            return r1;
        } else {
            r1.parent = r2;
            if (r1.rank == r2.rank) //preserve Rank for new root
                r2.rank++;
            return r2;
        }
    }
}

```

Correctness Highlights: By the correctness of *findRepresentative*, $r1$ is the root of the tree holding x , and $r2$ is the root of the tree holding the node on which this method is called. Clearly when $r1 = r2$ the correct value is returned. We now focus on when $r1 \neq r2$. Consider the following subcases:

$r1.rank > r2.rank$: In this case $r2$ becomes a child of $r1$. The new *height*($r1$) can be no greater than the maximum of the old *height*($r1$) and $1 + \text{height}(r2)$. However since the old rank of $r1$ is strictly greater than the rank of $r2$, we know by ROOTRANK, that this height maximum is bounded by the rank of $r1$, which is not changed. Therefore, ROOTRANK is preserved. The only parent reference to change is that the parent for $r2$ is changed to $r1$. Since $r1.rank > r2.rank$, PARENT RANK is preserved. The parent of $r1$ is not changed and so ROOT is preserved. Finally, since all elements in $T(r1)$ and $T(r2)$ will only be in the new component, PARTITION is preserved.

$r1.rank < r2.rank$: This case is symmetric to the first. While the code has one extra conditional, its body will not be executed in this case.

$r1.rank = r2.rank$: Observe that $r2$ will become the new root of the component which contains all elements in $T(r1)$ and $T(r2)$. So PARTITION is preserved. By PARENT RANK, prior to making $r1$ the child of $r2$, the height of $r1$ is no greater than $r1.rank$, and the height of $r2$ is no greater than $r2.rank$. Thus, once $r1$ becomes a child of $r2$, the height of $r2$ increases by at most one. So incrementing $r2.rank$ preserves ROOTRANK. Also, by incrementing $r2.rank$ it ensures that the rank of $r2$ is strictly larger than that of its children, preserving PARENT RANK. The parent of $r2$ is not changed, so ROOT is preserved. ■

6.7 Performance Analysis

The asymptotic time complexities of all public methods for the union-find data structure are shown in Table 6.4. It is easily seen that the worst-case time complexity for *get*, *set*, and *toString*, which only apply to a single partition element, are constant. The rest of the section focuses on the *findRepresentative*, *sameComponent*, and *union* methods.

To state the bounds, we must first define the **Ackermann function** [1]. We use the definition provided by Tarjan [146]. (Another commonly used definition is that given by Buck [36].) Ackermann's function $A_i(j)$ can be defined recursively as:

$$A_i(j) = \begin{cases} 2^j & \text{for } i = 1 \\ A_{i-1}(2) & \text{for } j = 1 \text{ and } i \geq 2 \\ A_{i-1}(A_i(j-1)) & \text{for } i, j \geq 2. \end{cases}$$

Observe that $A_4(n)$ yields the sequence $\langle 1, 2, 4, 16, 65536, 2^{65536}, \dots \rangle$. The **inverse Ackermann function** is defined as:

$$\alpha(m, n) = \min\{i \bmod A_i(\lfloor m/n \rfloor) > \log_2 n\}.$$

For all practical values of n and m , $\alpha(m, n) \leq 4$. So the inverse Ackermann function can be thought of as a constant that is at most 4. However technically, as m and n go to infinity, so does $\alpha(m, n)$.

method	complexity
constructor	$O(1)$, amortized
<i>get()</i>	$O(1)$, worst-case
<i>set(o)</i>	$O(1)$, worst-case
<i>toString()</i>	$O(1)$, worst-case
<i>findRepresentative(x)</i>	$O(\alpha(m, n))$, amortized
<i>sameComponent(x)</i>	$O(\alpha(m, n))$, amortized
<i>union(x)</i>	$O(\alpha(m, n))$, amortized

Table 6.4 Summary of the asymptotic time complexities for the union-find data structure where n is the number of calls to the constructor, and m is the number of calls to either *findRepresentative* or *union*. Note that $\alpha(n, m) \leq 4$ for all values of n and m that would ever occur. Recall that *get*, *set*, and *toString* only apply to a single node of the union-find data structure.

Tarjan [144, 145] proved that when using both union by rank and path compression, that the amortized time complexities given in Table 6.4 hold. Thus, the worst-case running time of a sequence of m operations, n of which are calls to the constructor (to make a new partition element) is $O(m\alpha(m, n))$, which is optimal in that any pointer-based data structure requires $\Omega(m\alpha(m, n))$ time to perform these operations. So the amortized cost for each operation is $\Omega(\alpha(m, n))$.

While we do not provide the full analysis here, we cover some of the key properties about the rank that are used in the proof. First we inductively prove that $|T(x)| \geq 2^{x.rank}$. If x is a singleton, the rank is 0. Thus it holds that $1 \geq 2^0$. We now consider when y becomes a child of x during a union. If $x.rank > y.rank$ then the rank of x does not change while the size of $T(x)$ increases, thus the claim still holds. Finally, we consider when $x.rank = y.rank$. Let r be the rank of x before the union, and let r' be the rank of x after the union. By the inductive hypothesis, before the merge $|T(x)| \geq 2^r$ and $|T(y)| \geq 2^r$. Thus after the union is completed $|T(x)| \geq 2 \cdot 2^r = 2^{r+1} = 2^{r'}$ since $r' = r + 1$. This completes the inductive proof that $|T(x)| \geq 2^{x.rank}$.

We now argue that for any integer $k \geq 0$, the number of nodes of rank k is at most $n/2^k$. When any node x is assigned a rank of k , $|T(x)| \geq 2^k$. Furthermore, by PARENTRANK all of the descendants of x have a rank less than k , and their rank never changes. Since there are at most n nodes and at least 2^k nodes must be in a subtree of a node assigned a rank of k , it follows that at most $n/2^k$ nodes are ever assigned rank k by the algorithm. As a result of this claim it immediately follows that for all x , $x.rank \leq \lceil \log_2 n \rceil$.

The $O(\log n)$ upper bound on the height of any tree in the union-find forest immediately yields an $O(\log n)$ worst-case bound for *findRepresentative* and *sameComponent*. The time complexity for *union* is constant. These upper bounds do rely upon union by rank but apply regardless of whether or not path compression is used.

To obtain the stronger amortized $O(\alpha(m, n))$ complexity bound [144], path compression is needed. We refer the reader to Cormen et al. [42] or Tarjan [146] for the analysis of the amortized complexity when using union by rank.

6.8 Case Study: Preserving Locators When Merging Data Structures

There are many pointer-based data structures for which a merge of two data structure instances can be implemented in either constant or logarithmic time. For example, the singly linked list

(Chapter 15), the doubly linked list (Chapter 16), the leftist heap (Chapter 26), the pairing heap (Chapter 27), and the Fibonacci heap (Chapter 28) have this property.

Let d_1 and d_2 be the two data structure instances that are being merged. We assume that these are pointer-based data structures, each composed of static objects of type T . We refer to each object of type T as a **node**. We use $|d_1|$ to denote the number of nodes in d_1 and $|d_2|$ to denote the number of nodes in d_2 . For ease of exposition, we assume that $|d_1| > |d_2|$.

Before discussing the merging of data structures, it is important to recall that a data structure's nodes, and its other affiliated objects like locators, are typically implemented as inner classes or static nested classes. Each instance of an inner class contains an implicit reference (added by the compiler) to the instance of the outer class with which it is affiliated. For example, a locator may be implemented as a member class so that it can (implicitly) refer to its data structure's modification count (Section 5.8.5) or to call its data structure's *remove* method. In contrast, an instance of a static nested class does not have an implicit reference to an instance of the outer class. Instead, instances of static classes stand on their own, and any reference to a particular instance of the outer class must be explicitly declared by the programmer.

When we merge two data structures, we would prefer not to make copies of the nodes, but instead to reuse the existing nodes to save time and space. Furthermore, it is infeasible to make copies of the locators since the application program already has references to the original ones. Therefore, when we merge two data structures, their nodes and locators must change affiliation. A node or locator may belong to one data structure before the merge, and belong to a different data structure after the merge. In other words, each of these nodes and locators must now refer to the correct (new) data structure containing all of the elements. If the node or locator is an instance of a static nested class, then a mechanism is needed to update its explicit reference. However, if the node or locator is an instance of an inner class, then there is no way to change the implicit reference because it is automatically passed by the compiler as an implicit parameter to the constructor and assigned to a hidden variable to which future assignment is not possible. So, to support merging of data structures with object re-use, it is necessary for any inner classes to become static, and to replace the ordinarily implicit references to the containing object by explicit references that can be reassigned.

Now, when the merge takes place, we do not want to incur the linear time overhead that would result from updating an explicit reference to the data structure in every node. Moreover, such an update to each locator would not even be possible since the data structure implementation typically does not retain references to its locators. For these reasons, our solution to merging data structures will involve a level of indirection of these references. Each instance of a node or locator requiring a reference to its data structure will instead refer to a proxy that contains the actual reference. For each data structure, we will create only one such proxy object, shared by all of its nodes and locators. In this way, when two data structures are merged, changing a single reference within the proxy will suffice to update all of the affiliated nodes and locators at once.

This proxy arrangement solves the affiliation problem when merging two data structures, but one problem remains. Suppose the data structure resulting from this merge is then merged with another data structure, and then this result is merged, yet again, with another. With each merge we accumulate more and more proxies, and all of them would need to be updated so that the original nodes and locators have the proper affiliation. Fixing up more and more proxies at each merge is clearly unacceptable, because after enough merges, with elements being added and removed, the number of proxies could even exceed the size of the collection! This is where union-find comes to the rescue.

We now describe how to use the union-find data structure to support nearly constant-time merging of two data structure instances. For each data structure instance, we create a single partition element, whose application data is the data structure itself. This partition element serves as the proxy shared by all the nodes and locators of the affiliated data structure that require such a reference. This proxy is also retained in an instance variable of the data structure for the purpose of initializing each new node and locator affiliated with that data structure. Whenever one of these affiliated objects needs

a reference to its (current) data structure, to access a field or call a method, it uses as the required reference the application data from the representative element of its proxy. Because every partition element is initially in a component by itself, when $d1$ is first created, its partition element is its own representative, so each affiliated object will find $d1$ as its data structure, since $d1$ is the application data for its partition element. A similar situation applies to $d2$, and any other newly created data structure.

The merge of $d1$ and $d2$ to produce a resulting data structure d is carried out in four steps.

1. A fresh data structure instance d is created and is populated with the nodes from $d1$ and $d2$, using a merge algorithm customized for that type of data structure. Clearly, $d.size$ will be $d1.size + d2.size$, but the merge itself need not take linear time. In fact, it may be as simple as linking the tail of one list to the head of another.
2. Two union operations must be performed. A union operation is performed on the proxies of $d1$ and $d2$ so that they are in the same component. Then, a union is performed on the proxies of $d1$ and d , so that now all three proxies are in the same component.
3. The application data of the representative element of the component is assigned to be d , so that now all nodes and locators in both $d1$ and $d2$ will be affiliated with the new data structure. The application data in the other two proxies is set to null, so that no references to the old data structures remain in the component to prevent garbage collection.
4. The original data structures $d1$ and $d2$ are reinitialized with a size of 0 and with new proxies. At this point, these data structures are empty and independent, and can be either reused or discarded.

Figures 6.5, 6.6, and 6.7 show the result of merging three data structures. Figure 6.5 shows the three data structures prior to the merge. Each has its own union-find node (the proxy) and a sample locator to illustrate the relationships. Note that each union-find node is its own representative, indicated by the self loop of the parent reference. Also, note that each proxy has its corresponding data structure as its application data. Merging all three data structures results in the relationships shown in Figure 6.6. Note that all of the old proxies (shown shaded) belong to the same component, so all of the old locators would now see d as the application data for their proxies. In addition, note that $d1$, $d2$, and $d3$ all have new proxies that would be used to initialize any nodes or locators created for those (now empty) collections. Figure 6.7 shows what happens as new affiliated objects are added to each data structure. Note that all of the old locators still refer to the old proxies, which all belong to the single component whose representative refers to d . But each of the four data structures has a life of its own. For example, new items created $d1$ are affiliated with $d1$ as expected, and new items in d are affiliated with that structure, also as expected. If desired, the data structures containing the new items could be merged again later to create another new collection.

A complete implementation of this approach to merging data structures is provided for the leftist heap (Chapter 26). The same approach could be applied in a straightforward manner to any of the other data structures, resulting in an efficient object-oriented merge implementation that reuses the objects in the internal representation.

6.9 Further Reading

The union-find data structure is due to Robert Tarjan [144, 145]. Cormen et al. [42] and Tarjan [146] are good sources for the analysis of the amortized complexity of the union-find methods. The first

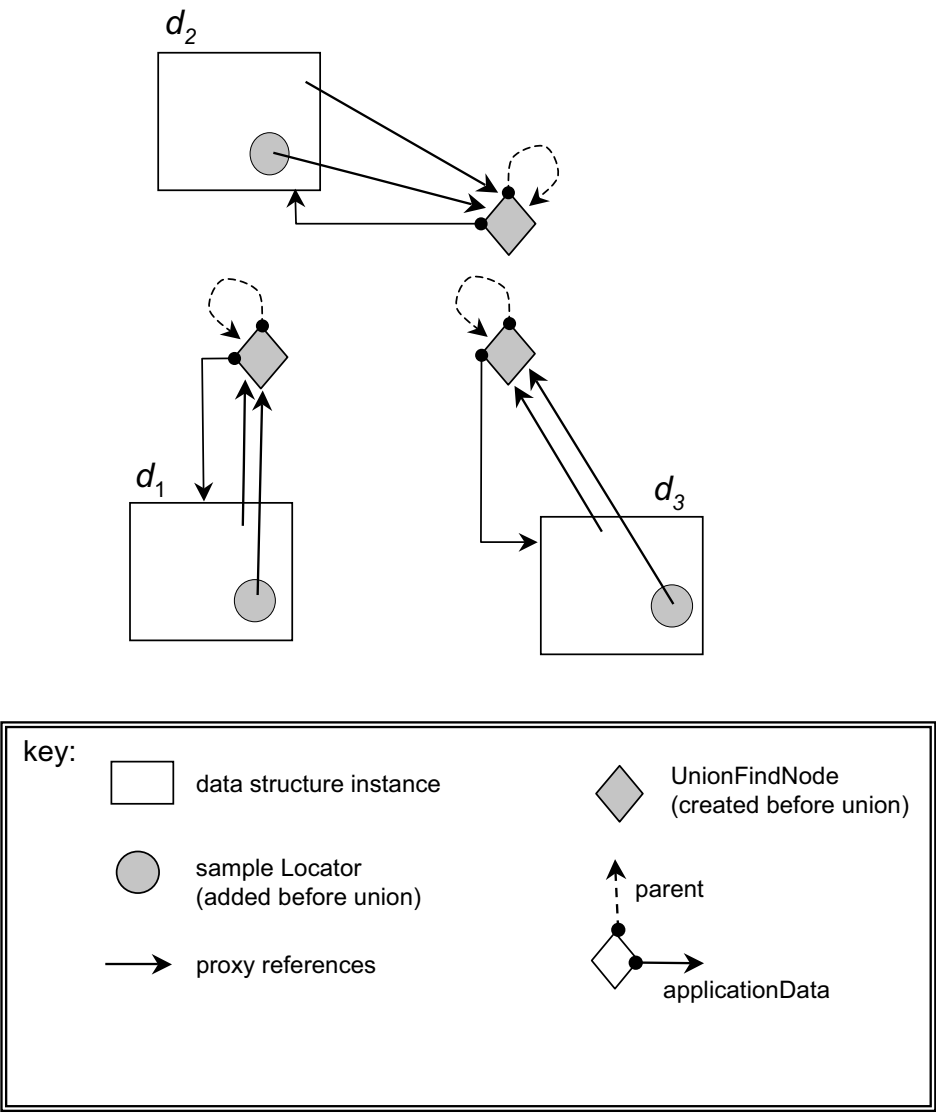


Figure 6.5
Independent data structures and their proxies prior to a merge.

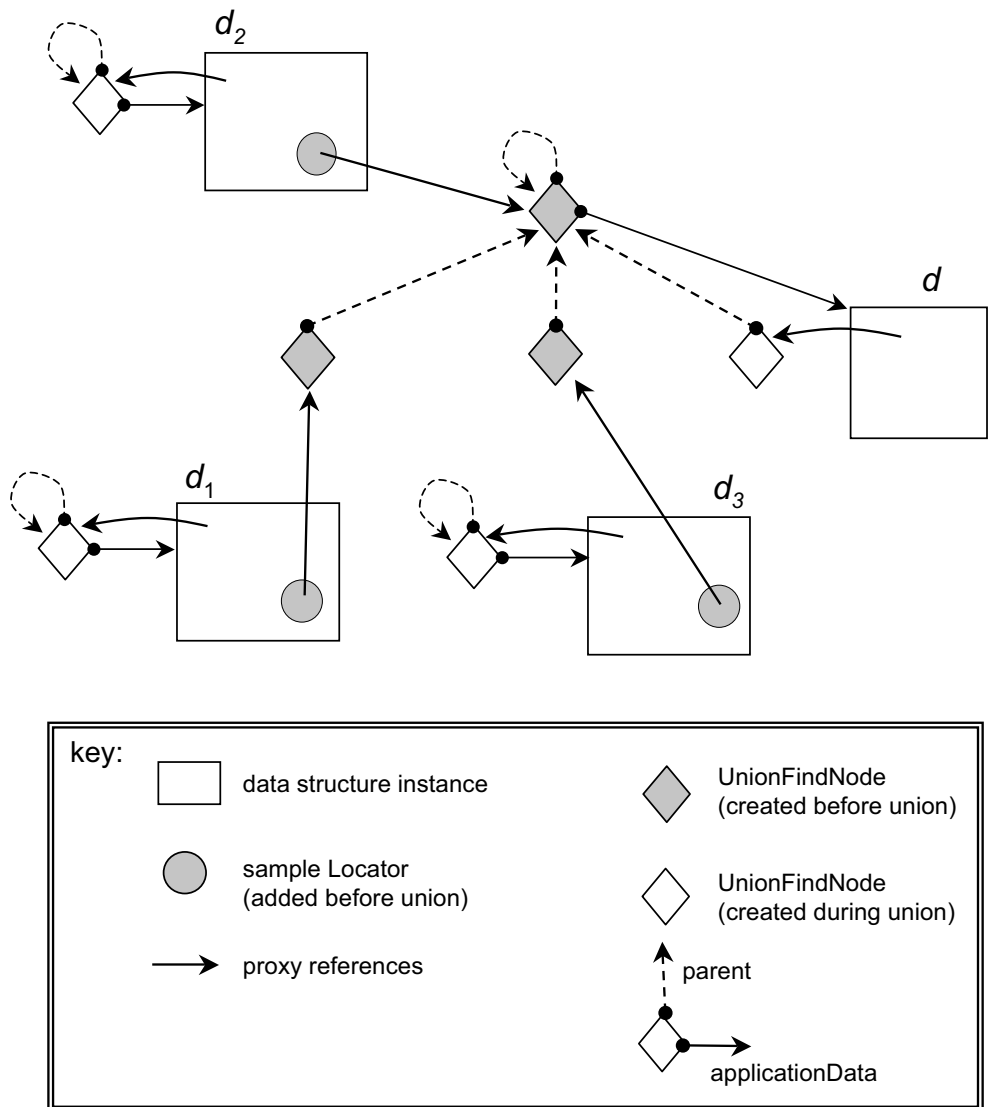


Figure 6.6

Resulting data structure and proxies after the merge. The old proxies share a single representative, which refers to d .

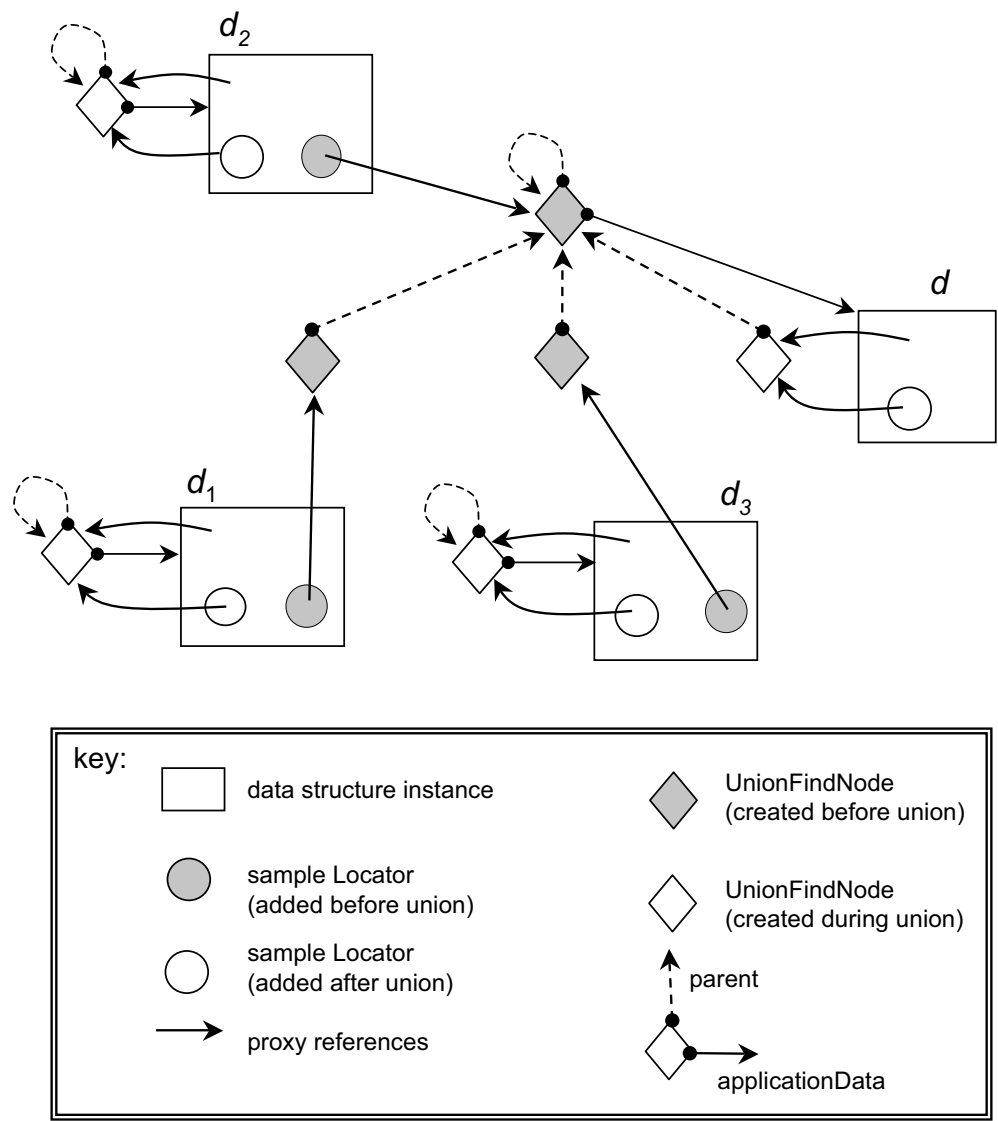


Figure 6.7
New locators refer to new proxies after the merge, while old locators continue to refer to their old proxies.

tight bound on the amortized complexity of the *findRepresentative* and *union* methods in terms of the inverse Ackermann function was given by Tarjan [143, 144]. Earlier work showed a similar result in terms of the **iterated logarithm** function which is also a very slow growing function but not as slow growing as the inverse Ackermann function [3, 85]. An alternate analysis that yields an amortized cost for a function very similar to the inverse Ackermann function can be found in Cormen et al. [42]. Tarjan [145] and Fredman and Saks [63] showed a lower bound proving that the union-find data structure performs optimally under certain conditions. However, for some applications, the Partition ADT can be implemented so all methods have constant amortized cost [69].

The *findRepresentative* method is a **two-pass method**. It makes one pass up the tree to find the root, and then it makes a second pass down the path to update each node so that its parent reference directly references the root. Tarjan and van Leeuwen [147] present a one-pass method which sometimes improves performance by a constant factor.

6.10 Quick Method Reference

UnionFindNode Public Methods

- p. 77** *UnionFindNode*(T data)
- p. 78** *UnionFindNode*<T> **findRepresentative**()
- p. 77** T **get**()
- p. 78** boolean **sameComponent**(PartitionElement<T> x)
- p. 77** void **set**(T data)
- p. 77** String **toString**()
- p. 79** PartitionElement<T> **union**(PartitionElement<T> x)

Chapter 7

Collection of Elements

A wide variety of data structures maintain a collection of elements. While these data structures vary in how they manage their elements (in terms of position, logical organization, etc.), there are certain operations that all collections support. These methods comprise the *Collection* interface defined and discussed in this chapter. Section 7.1 describes the general semantics of each method in the *Collection* interface, and Section 7.2 describes a *Tracked* extension of this interface to support trackers. Section 7.3 lists ADTs that support the *Collection* interface. Some of these impose additional semantics on methods in the *Collection* interface, and some provide additional specific methods.

Many applications need to iterate over a collection, and some need to keep track of the location of a particular element to efficiently find or remove the element at a later point. To support such applications, we introduce, in Section 5.8, the notion of a **Locator**, whose interface extends Java's *Iterator* interface. We described two kinds of locators: markers, which mark a particular location in a positional collection, and trackers, which track a particular element held in the collection. All collections provide an *iterator* method for iterating over the collection. The iterator method always returns either a tracker or a marker, depending upon whether or not the implementation is tracked.

Terminology: The following definitions facilitate discussion of the data structures that implement the *Collection* interface. We use many of these terms throughout Part II.

- We let n denote the number of elements in the collection.
- We use the **target** as the element that is sought in a search, within methods like *contains* or *remove*.
- Since Java does not support pointers on which arithmetic operations can be performed, we usually refer to a variable that contains a memory address as a **reference**. However, we will use the term **pointer-based data structure** when the internal representation consists of many individual objects linked together by references.
- We say that implementations that support trackers are **tracked** implementations, whereas those that only support markers are **untracked** implementations.
- A collection implementation is said to be **oversized** if it is implemented using an underlying data structure that may at times have empty space. For example, if a collection's internal representation is an array of length 8 that may hold fewer than 8 elements, its implementation is oversized.
- A collection implementation is said to be **elastic** if its space usage is always directly proportional to the number of elements stored. For example, if a collection's internal representation is a list structure that always has exactly one list item per element, its implementation is elastic.

- When there is no limit (excluding the bounds constrained by the memory of the computer) on the number of elements held in a collection, we say the ADT is **unbounded**.
- Similarly, when there is a user-provided limit on the number of elements that can be held in the collection, we say that the ADT is **bounded**.
- An *AtCapacityException* is thrown if an insertion is attempted on a bounded collection that is already at capacity.
- We say that two references are **equivalent** if and only if they are equal (which includes both being *null*) or if they refer to elements that are equal when using the *equals* method to test for equality, or alternatively if they refer to objects that are deemed equal by an application-provided equivalence tester. Since the need to test two elements for equivalence is so common, we provide a public abstract class *Objects* with a static boolean *equivalent* method that tests if two references are equivalent. (See Section 5.2.2.) When *o1.equivalent(o2)* is *true*, we denote this as $o1 \cong o2$.
- For ease of exposition, we define **FORE** to be logically just before the first element in the collection, and **AFT** to be logically just after the last element in the collection.
- Recall that when a locator is being used for iteration, we define a **persistent element** as an element that was in the collection when the locator was first initialized to FORE, and remains in the collection throughout the use of the locator.

7.1 Collection Interface

package collection

Iterable<*E*>

└ *Collection*<*E*>

The *Collection* interface includes the following methods.

void accept(Visitor<? super E> v): Traverses each element of this collection, in the iteration order, on behalf of the visitor.

void add(E value): Inserts *value* into the collection in an arbitrary location. If a tracker is to be returned then the method *addTracked* which is part of the *Tracked* interface should be called instead. An *AtCapacityException* (an unchecked exception) is thrown when a bounded collection is already at capacity.

void addAll(Collection<? extends E> c): Adds all elements in *c* to this collection.

void clear(): Removes all elements from this collection.

boolean contains(E target): Returns *true* if an element equivalent to *target* exists in this collection. Otherwise *false* is returned.

void ensureCapacity(int capacity): Increases the capacity of this collection, if necessary, to ensure that it can hold at least *capacity* elements. For elastic implementations, this method does nothing.

int getCapacity(): Returns the current capacity of this collection. For elastic implementations *Integer.MAX_VALUE* is returned.

Comparator<E> getComparator(): Returns the comparator for elements in this collection.

E getEquivalentElement(E target): Returns an element in the collection that is equivalent to *target*. It throws a *NoSuchElementException* when there is no equivalent element. The *contains* method should be used to determine if an element is in the collection.

Locator<E> getLocator(E target): Returns a locator that has been initialized to an element equivalent to *target*. Like the *iterator* method, this method enables navigation, but from a specified starting point. This method throws a *NoSuchElementException* if there is no equivalent element in the collection.

int getSize(): Returns the number of elements, *n*, in this collection.

boolean isEmpty(): Returns true if this collection contains no elements, and otherwise returns false.

Locator<E> iterator(): Returns a locator that has been initialized to FORE. The locator returned can be used to navigate within the collection and to remove the element currently associated with the locator. In general, no assumption is made about the **iteration order**, the order in which the iterator advances through the collection. However, some collection ADTs specify a particular iteration order.

boolean remove(E target): Removes from this collection an arbitrary element equivalent to *target*, if such an element exists in the collection. It returns *true* if an element was removed, and *false* otherwise.

void retainAll(Collection<E> c): Removes from this collection all elements for which there is no equivalent element in *c*. Thus, the elements that remain are those in the intersection of *c* and this collection.

Object[] toArray(): Returns a Java primitive array of length *n* that holds the elements in this collection in the iteration order.

void toArray(E[] a): Fills a Java array with the elements in this collection in the iteration order, and returns the array that was filled. If the array provided as a parameter is not large enough to hold all the elements of the collection, a new array of the same type is created with length *n* and is returned instead of the provided one.

String toString(): Returns a string that includes each element in this collection (as produced by the *toString* method for that element), in the iteration order.

void trimToSize(): Trims the capacity of an oversized collection to exactly hold its current elements. An application can use this operation to minimize the space usage. For elastic implementations, this method does nothing.

Critical Mutators for Collection: *add* (except when *p = size*), *addFirst*, *addAll*, *clear*, *heapsort*, *insertionsort*, *mergesort*, *quicksort*, *radixsort*, *removeFirst*, *removeRange* (except when *toPosition = size - 1*), *repositionElementByRank*, *retainAll*, *swap*, *treesort*

7.2 Tracked Collection Interface

package collection

Collection<*E*>

↑ *Tracked*<*E*>

The *Tracked* interface adds a single method to the *Collection* interface that allows the user to obtain a tracker when an element is inserted. Any tracked data structure should implement this interface.

***Locator*<*E*> *addTracked*(*E o*):** Inserts *o* into the collection in an arbitrary location and returns a tracker to the inserted element. An *AtCapacityException* (an unchecked exception) is thrown if the collection is already at capacity.

Critical Mutators for TrackedCollection: All critical mutators of the *Collection* interface and *addTracked* (except when $p = \text{size}$)

7.3 ADTs Implementing the Collection Interface

This section summarizes the purpose of each of the collection ADTs. For detailed guidance on selecting among them, refer to Section 2.4.

7.3.1 Manually Positioned Collections

PositionalCollection - This is the most general manually positioned collection. The application can insert or remove an element at any specified position. Positions are, by definition, 0 to $n - 1$. Also, the iteration order is position 0, position 1, ..., position $n - 1$.

Buffer - This ADT is a specialization of the Positional Collection ADT in which the user can insert a new element only at the front or back of the collection. Similarly, an element can be removed only from the front or back. A buffer can be bounded or unbounded.

Queue - This is a specialization of the Buffer ADT in which the application program can insert elements only at the back of the line (called *enqueue*) and remove elements only from the front of the line (called *dequeue*). The iteration order is that of a first-in, first-out (FIFO) line. A queue can be bounded or unbounded.

Stack - This is a specialization of the Buffer ADT in which the application program can only insert (*push*) and remove (*pop*) elements at the front (often called the top) of the stack. The iteration order is that of a last-in, first-out (LIFO) line. A stack can be bounded or unbounded.

7.3.2 Algorithmically Positioned Untagged Collections

Set - The elements stored in a set are unique, but they need not be comparable. The primary methods are to add an element, remove an element, and to determine if a given element is contained in the set. Set data structures typically support these methods in expected constant time.

PriorityQueue - The elements stored in a priority queue must be comparable. The order defined by the comparator is intended to be based on some intrinsic priority among the elements where $e_1 > e_2$ implies that e_1 has a higher priority than e_2 . A priority queue is intended for applications in which efficient access is required for the highest priority element. While there is a total order defined over the elements, iteration need not follow this order. Also, data structures for a priority queue typically require linear time to search for a specific element. If efficient access of the lowest priority element is instead desired, a reverse comparator (Section 5.2.3) can be used to wrap the desired comparator.

OrderedCollection - This ADT is designed for applications for which the elements are uniquely ordered, and for which iteration must correspond to this unique order. Not as efficient as sets, most OrderedCollection implementations can add an element, remove an element, and find an element in logarithmic time. In addition, there are efficient methods to find the minimum element (first in the iteration order), maximum element (last in the iteration order), and to move forward or backwards in the iteration order. Thus, an application program can efficiently find all elements in a certain range (a **range search**), or find the element in the iteration order that is closest to a desired value.

DigitizedOrderedCollection - This ADT is a specialized OrderedCollection ADT in which a digitizer (Section 5.3) is provided for the elements. A total order over the elements is implicitly defined by the digitizer. This ADT supports all the methods of the OrderedCollection ADT. In addition, it provides methods to (1) find all the elements in the collection that share the longest common prefix with a given element, and (2) to find all elements in the collection that begin with a provided prefix. Most of the data structures we present for maintaining digitized ordered collections guarantee a worst-case search time proportional to the length of the prefix needed to distinguish the given element from all elements in the collection.

SpatialCollection - This ADT is designed for when the elements are multiply ordered, and the application must efficiently find the set of elements that are within a certain range in each ordering. For example, a collection of locations could be represented as longitude and latitude coordinates. One ordering is based on the longitude and a second ordering is based on the latitude. A spatial collection would support efficiently locating all elements in a rectangular geographic region. In general, multiply ordered elements need not truly be spatial as in this example, but many problems still reduce to finding a box in a d -dimensional space. For example, one could maintain a similar collection of locations that also has a time coordinate. The application program might want to find all civilizations that had a latitude in a given range, a longitude in a given range, and existed during a given time period. Geometrically, this corresponds to finding all locations (which can be viewed as a 3-dimensional point) that are in the box defined by the specified ranges. The iteration order for a spatial collection can be arbitrary.

7.3.3 Algorithmically Positioned Tagged Ungrouped Collections

Mapping - This ADT is the tagged version of the Set ADT. Each tag in a mapping is unique and is therefore called a **key**. The key is used only to identify the element, and an association is created between the key and the element. This ADT is sometimes called a **dictionary** since it is used to look up elements by their keys. However, unlike an alphabetized dictionary used to look up the meaning of English words, keys are not ordered by comparison.

TaggedPriorityQueue - This ADT is the tagged version of the PriorityQueue ADT. The tags must be comparable but need not be unique.

TaggedOrderedCollection - This ADT is the tagged version of the OrderedCollection ADT. Again, the tags must be comparable but need not be unique.

TaggedDigitizedOrderedCollection - This ADT is the tagged version of the DigitizedOrderedCollection ADT. The collection must use an application-defined digitizer that implicitly defines the ordering among the tags.

TaggedSpatialCollection - This ADT is the tagged version of the SpatialCollection ADT. The tags are multiply ordered and need not be unique.

7.3.4 Algorithmically Positioned Tagged Grouped Collections

BucketMapping - This ADT is the tagged bucket version of the Mapping ADT when the tag need not be unique. All elements with the same tag are grouped together in a bucket that can be quickly accessed via the tag. As with the Mapping ADT, the tag need not be comparable.

TaggedBucketPriorityQueue - This ADT is the tagged bucket version of the TaggedPriorityQueue ADT. The tags must be comparable but need not be unique.

TaggedBucketOrderedCollection - This ADT is the tagged bucket version of the TaggedOrderedCollection ADT. Again, the tags must be comparable but need not be unique.

TaggedBucketDigitizedOrderedCollection - This ADT is the tagged bucket version of the TaggedDigitizedOrderedCollection ADT. The collection must use an application-defined digitizer that implicitly defines the ordering among the tags.

TaggedBucketSpatialCollection - This ADT is the tagged bucket version of the TaggedSpatialCollection ADT. The tags are multiply ordered and need not be unique.

Chapter 8

Abstract Collection

package collection

AbstractCollection<E> implements *Collection*<E>

8.1 Internal Representation

While most methods in the *Collection* interface depend upon details of the internal representation, some methods can be implemented here and shared by all data structures that implement a collection. Some of these methods involve maintaining and accessing the number of elements in the collection (e.g., *size*, *isEmpty*), and others are implemented in terms of abstract methods like *add*, making use of the **template method design pattern**, as we do often throughout this book. In some cases, the implementations provided here are “brute-force,” implementations but they can be over-ridden when a more efficient algorithm exists for a specific data structure. Also, the methodology and associated instance variable *version*, for triggering a *ConcurrentModificationException* when appropriate, is included here to be inherited by all collection data structures.

Critical Mutators: *add*, *addAll*, *clear*, *ensureCapacity*, *remove*, *retainAll*, *trimToSize*

Instance Variables and Constants: For all oversized implementations, the internal representation must be initialized at a given capacity. For such implementations, the user can provide an initial capacity. Otherwise, the *DEFAULT_CAPACITY* is used. The instance variable, *size*, holds the number of elements in the collection. The variable, *version*, is an object that maintains the current modification count so that a *ConcurrentModificationException* can be thrown when necessary, as discussed in Section 5.8.5. Finally, *comp* is the comparator used to compare elements.

```
protected static final int FORE = -1;           //precedes position 0 (logically)
protected static final int NOT_FOUND = -2;      //return value for position if not found
public static final int DEFAULT_CAPACITY = 8;   //default capacity
protected int size = 0;                        //the number of elements, n, in the collection
protected Version version = new Version();      //keeps modification count
public Comparator<? super E> comp;             //for comparing user values

public AbstractCollection(Comparator<? super E> comparator) {
    comp = comparator;
}
```


8.2 Representation Properties

We introduce a single representation property that will be maintained by all data structures that implement a collection. Each specialized collection provides its own additional properties.

SIZE: The value of the instance variable *size* is always the number of elements held in the collection. That is, $size = n$.

8.3 Methods

In this section we present the internal and public methods of the `AbstractCollection` class.

8.3.1 Trivial Accessors

The following accessors are very straightforward. The method *isEmpty* returns *true* if and only if no elements are stored in the collection.

```
public boolean isEmpty() {
    return size == 0;
}
```

| *Correctness Highlights:* By SIZE, the collection is empty exactly when $n = 0$. ■

The method *getSize* returns the number of elements stored in the collection.

```
public int getSize() {
    return size;
}
```

| *Correctness Highlights:* Follows directly from SIZE. ■

Recall that *getCapacity* returns the current capacity of the collection. By default this method returns `Integer.MAX_VALUE`. For elastic implementations, no change is needed. However, this method must be overridden in oversized implementations.

```
public int getCapacity() {
    return Integer.MAX_VALUE;
}
```

The *getComparator* method returns the comparator used for this collection.

```
public Comparator<? super E> getComparator() {
    return comp;
}
```

The *compare* method takes *e1*, the first element to compare, and *e2*, the second element to compare. It returns a negative value when $e1 < e2$, zero when *e1* equals *e2*, and a positive value

when $e1 > e2$. As discussed in Section 5.2.2, an application can provide its own comparator as an argument to the constructor.

```
protected int compare(E e1, E e2) {
    return comp.compare(e1, e2);
}
```

To enable an application to define its own notion of equivalence, we include the *equivalent* method. This method takes *e1*, one element to compare, and *e2*, the second element to compare. It returns *true* if and only if *e1* and *e2* are equivalent with respect to the set.

```
protected boolean equivalent(E e1, E e2) {
    return comp.compare(e1, e2) == 0;
}
```

8.3.2 Algorithmic Accessors

The *toArray* method returns a Java primitive array of capacity *n* that holds each element of the collection.

```
public Object[] toArray() {
    Object[] result = new Object[getSize()];
    int i = 0;
    for (E e: this)
        result[i++] = e;
    return result;
}
```

A second *toArray* method takes as its parameter *array*, an array of the correct type into which the collection's elements are to be placed. It returns a Java primitive array of capacity at least *n* that holds each element of the collection. The elements are placed in the array, beginning at index 0, in the iteration order of the collection. If the provided array is not large enough, a new array of length *n* is created and returned instead of the provided one. In this case, reflection is used so that the new array is the same type as the one provided.

```
public E[] toArray(E[] array) {
    if (array.length < size)
        array = (E[]) java.lang.reflect.Array.newInstance(
            array.getClass().getComponentType(), size);
    int i = 0;
    for (E e: this)
        array[i++] = e;
    return array;
}
```

Recall that *contains* takes *value*, the element to be located, and returns *true* if and only if an equivalent element exists in the collection. This implementation for *contains* takes linear time, so it is overridden by a more efficient method in most data structure implementations.

```

public boolean contains(E value) {
    for (E e : this)
        if (equivalent(e, value))
            return true;
    return false;
}

```

The *getEquivalentElement* method takes *target*, the target element, and returns an equivalent element that is in the collection. It throws a *NoSuchElementException* when there is no equivalent element in the collection.

```

public E getEquivalentElement (E target) {
    Locator<E> loc = iterator();
    while (loc.advance()) {
        if (equivalent(loc.get(), target))
            return loc.get();
    }
    throw new NoSuchElementException();
}

```

We introduce an internal method *writeElements* that takes *s*, a string builder, and appends, to the string buffer a comma-separated string representation for each element in the collection, in the iteration order. This method relies on the *toString* method for the elements stored within the collection. For the sake of efficiency, *StringBuilder* is used instead of *String*. If a *String* were directly used, then this method (and consequently the collection's *toString* method) would take quadratic time since the accumulated string would be copied for each append. (The locator *hasNext* and *next* could have been used, as is typically done when using a Java iterator.)

```

protected void writeElements(StringBuilder s) {
    Locator<E> loc = iterator();
    while (loc.advance()) {
        s.append(loc.get());
        if (loc.hasNext())
            s.append(", ");
    }
}

```

The *toString* method returns a comma-separated string that shows the sequence held in the collection, in the iteration order. Braces mark the beginning and the end of the collection.

```

public String toString() {
    StringBuilder s = new StringBuilder("{}");
    writeElements(s);
    s.append("{}");
    return s.toString();
}

```

The *accept* method takes *v*, a visitor. It traverses the entire collection on behalf of a visitor. It throws a *VisitAbortedException* when the traversal is aborted due to an exception raised by the visitor, in which case the cause held by the *VisitAbortedException* is the exception thrown by the visitor.

```

public final void accept(Visitor<? super E> v) {
    try {
        traverseForVisitor(v);
    } catch (Throwable cause) {
        throw new VisitAbortedException(cause);
    }
}

```

The *traverseForVisitor* method takes *v*, a visitor. It traverses the collection applying *v* to each element. Many data structures override this method with a more efficient implementation than this one. In particular, one may override this method to provide natural and efficient recursive traversal of a data structure on behalf of a visitor.

```

protected void traverseForVisitor(Visitor<? super E> v) throws Exception {
    for (E e : this)
        v.visit(e);
}

```

One static *getElementAtRank* method that takes *coll*, the collection to use, and *rank*, the rank of the desired element. It returns the element that would be in position *r* if the collection were sorted, where the smallest element is defined as rank 0. The natural ordering of the elements is used for comparison. This is a non-mutating method.

```

public static<T> T getElementAtRank(Collection<T> coll, int rank) {
    return getElementAtRank(coll, rank, Objects.DEFAULT_COMPARATOR);
}

```

A second static *getElementAtRank* method takes *coll*, the collection to use, *rank*, the rank of the desired element, *comp*, the comparator that defines the ordering of the elements, and returns the element that would be in position *r* if the collection were sorted. This is a non-mutating method.

```

public static<T> T getElementAtRank(Collection<T> coll, int rank,
                                   Comparator<? super T> comp) {
    if (coll instanceof OrderedCollection)
        return ((OrderedCollection<T>) coll).get(rank);
    Array<T> a = new Array<T>(coll.getSize());
    a.addAll(coll);
    return a.repositionElementByRank(rank, comp);
}

```

8.3.3 Representation Mutators

Recall that *ensureCapacity* takes *capacity*, the desired capacity for the collection, and increases the capacity of the collection, if necessary, to ensure that it can hold at least *capacity* elements. This method must be overridden when some action is required to ensure the desired capacity.

```

public void ensureCapacity(int capacity) { }

```

The *trimToSize* method trims the capacity of this collection to be its current size. An application can use this operation to minimize space usage. This method must also be overridden for oversized implementations.

```
public void trimToSize() { }
```

Correctness Highlights: By definition, for elastic implementations, neither of these methods must perform any action. ■

8.3.4 Content Mutators

The *addAll* method takes *c*, the collection to be added. This method adds all elements in *c* to the collection. This method iterates through *c* and adds each element in *c* to the collection.

```
public void addAll(Collection<? extends E> c) {
    for (E e : c)
        add(e);
}
```

The *retainAll* method takes *c*, a collection, and updates the current collection to contain only elements that are also in *c*. In other words, the current collection is updated to be the intersection between itself and *c*. This method iterates through the collection and checks if each element is contained in *c*. Any element not contained in *c* is removed.

```
public void retainAll(Collection<E> c) {
    Locator<E> loc = iterator();
    while (loc.advance())
        if (!c.contains(loc.get()))
            loc.remove();
}
```

The *clear* method removes all items from the collection. This method uses an iterator and the *Locator remove* method. In many cases, a more efficient implementation exists, so this method will often be overridden.

```
public void clear() {
    Locator<E> loc = iterator();
    while (loc.advance())
        loc.remove();
}
```

8.4 Abstract Locator Inner Class

AbstractLocator<**T extends E**> implements *Locator*<*E*>

Just as we define an *AbstractCollection* class, we define an *AbstractLocator* class to provide a starting point for more specific *Locator* implementations. This inner class holds the methods related to detecting and handling concurrent modifications as discussed in Section 5.8.5.

This abstract class has a single instance variable *versionNumber* that stores the modification count for which the *Locator* is not stale.

```

protected int versionNumber;                                     //version number for locator

protected AbstractLocator() {
    versionNumber = version.getCount();
}

```

The *checkValidity* method compares the version number held by the locator with the current version number of the data structure object. It throws a *ConcurrentModificationException* when the locator version number is less than the modification count for the data structure object.

```

protected void checkValidity() {
    version.check(versionNumber);
}

```

The *updateVersion* method updates the version number for the locator to be the current modification count for the data structure object. This method is typically used whenever a modification is made from within the locator and thus it should not be invalidated.

```

protected final void updateVersion() {
    versionNumber = Math.max(versionNumber, version.getCount());
}

```

The *ignoreConcurrentModifications* method takes *ignore*, a boolean flag indicating if concurrent modification exceptions should be disabled. It sets the version number of the locator to *MAX_VALUE* when *ignore* is *true*, preventing concurrent modification exceptions from occurring. When *ignore* is *false*, it restores the version number to the smaller of the locator's version number and the current modification count for the data structure object. When the parameter is *true*, this method is used to suppress notification of concurrent modifications, possibly temporarily, so that iteration can continue even if the iteration order may not be consistent (elements may be visited twice, skipped, or seen out of order).

```

public void ignoreConcurrentModifications(boolean ignore) {
    if (ignore)
        versionNumber = Integer.MAX_VALUE;
    else
        versionNumber = Math.min(versionNumber, version.getCount());
}

```

The *ignorePriorConcurrentModifications* method resets the version number of the locator to the current modification count for the data structure object. This method allows the locator to continue iteration from this point, regardless of prior critical modifications of the data structure. Ignoring its history, the iterator will observe consistent iteration beginning at its current position in the collection. Calls to critical mutators in the future may induce a *ConcurrentModificationException* from this locator.

```

public void ignorePriorConcurrentModifications() {
    updateVersion();
}

```

We also support the *next* method of the Java *Iterator* interface, which moves the locator forward to the next element in the collection. It returns the element stored at the position to which the locator is moved. It throws an *AtBoundaryException* when the locator is already at AFT, and throws a *NoSuchElementException* when the locator is at the last element in the collection.

```

public E next() {
    advance();
    return get();
}

```

The other locator methods are left as abstract methods since they depend upon the internal representation of the data structure and its locator inner class.

8.5 Visiting Iterator

package collection

AbstractCollection<E>.AbstractLocator<E> implements *Locator<E>*

↳ **VisitingIterator** implements *Visitor<E>*, *Locator<E>*, *Runnable*

When traversal of a data structure is most naturally accomplished recursively, it is generally easiest to express that recursive algorithm directly and support traversal through the use of a visitor. However, implementing an iterator that follows the same iteration order as the natural recursive solution can be significantly more involved. Such implementations may involve keeping careful history information, or even a stack of visited items, as part of the state of the iterator, simulating the kind of information that would normally be kept within the execution stack during a recursive traversal of the data structure.

The following visiting iterator class provides a general way to create an iterator for any data structure that supports a visitor. It provides iteration over the collection by means of a (possibly recursive) traversal algorithm provided for visitors, without the need to define a customized iterator class for each such data structure. The visiting iterator works by creating a thread that uses the visiting algorithm of the data structure to traverse each element. In particular, to support visiting the elements one by one, the visiting iterator thread blocks after visiting each element and waits for that element to be consumed by a call to the iterator. This thread's execution stack fully captures the state of the (possibly recursive) traversal algorithm, and avoids the need to encode that state in an *ad hoc* manner within the state of the iterator itself. Of course, it would never make sense to use a visiting iterator when the visiting algorithm itself is defined in terms of an iterator, as in the case of the default *traverseForVisitor* implementation in the AbstractCollection class. Example uses of the visiting iterator are provided in our implementation of the leftist heap data structure (Chapter 26) and the quad tree data structure (Chapter 48).

Caveat: Because it executes in a separate thread and makes heavy use of thread synchronization, the visiting iterator is considerably less efficient than a typical custom iterator. The visiting iterator does provide a general-purpose iteration mechanism for recursive visiting patterns, and it is an interesting illustration of concurrency control and thread synchronization, but it is an option of last resort that should be used only when (1) a traversal algorithm for a visitor is easily defined, (2) defining an equivalent custom iterator implementation would be too complicated or require too much extra space within the data structure, and (3) iteration is performed rarely by the application. If the application performs iteration frequently but the data structure does not support iteration well, then it would be wise to consider replacing the use of iterators in the application by writing a custom visitor, or selecting an alternate data structure that does support an efficient iteration. Also note also that the thread created by the visiting iterator does not exit unless either the iteration order

is exhausted or the iterator's *cancel* method is called. It is essential that one of these two occur; otherwise, the visiting iterator thread will continue to exist in a blocked state.

The thread terminates after all items in the iteration order have been consumed using *next*, followed by a last call to *hasNext* to detect termination. Alternatively, one may call the visiting iterator's *cancel* method to stop iteration early and terminate the thread.

The visiting iterator provides only simple iteration, supported by the methods *hasNext* and *next*. The visiting iterator includes four instance variables. The boolean *hasItem* is *true* exactly when *nextItem* contains the next item and it has not yet been returned to the application in response to a call to the *next* method. The boolean *finished* is set to *false* when the iteration begins and remains *false* until the last element in the collection has been reached. Similarly, the boolean *canceled* is set to *false* when the iteration begins and remains *false* until the user requests that iteration be aborted.

```
boolean hasItem = false;
E nextItem = null;
boolean finished = false;
boolean canceled = false;
```

The constructor starts the visiting iterator thread.

```
public VisitingIterator() {
    (new Thread(this)).start();           //start the visitor thread
}
```

The synchronized *run* method, which is called when the thread is started, calls the *accept* method on the collection provided to the constructor. The boolean *finished* (which was initially *false*) is set to *true* only when *accept* completes.

```
public synchronized void run() {
    try {
        AbstractCollection.this.accept(this);           //begin accept method
    } catch (VisitAbortedException vae) {
        //iteration canceled
    }
    finished = true;                                     //only finished when it returns
    notify();
}
```

The synchronized *cancel* method aborts iteration to terminate the visiting iterator thread when iteration to completion is not required. Subsequent calls to *hasNext* will return *false*.

```
public synchronized void cancel() {
    canceled = true;
    notify();
}
```

The synchronized *visit* method takes *item*, the element to visit. Consider the situation in which the visiting algorithm is recursive. The visiting iterator thread runs until the next item to visit is reached, at which point the *visit* method is called for that item. At this point that item is stored in *nextItem* and *hasItem* is set to *true* to reflect that the iterator is holding the item to be returned from *next*. The thread then blocks, waiting until *next* is called to consume the item. If the iterator has been canceled, the next call to the *visit* method throws an exception to terminate traversal of the data structure.


```

public synchronized void visit(E item) throws Exception {
    if (canceled)
        throw new Exception("further iteration canceled");
    nextItem = item;                                //store next item
    hasItem = true;                                //item not yet consumed by application thread
    notify();                                       //release lock for hasNext
    try {
        wait();                                    //until notify from next
    } catch (InterruptedException ie) {}
}

```

The synchronized *hasNext* method returns *true* if and only if iteration has not reached the last element. If *hasItem* is *true* then the element stored in *nextItem* has not yet been returned, so *hasNext* can return *true*. Similarly, if *finished* is *true* then there is another item exactly when the item referenced by *nextItem* has not been returned, which is the case exactly when *hasItem* is *true*. If *hasItem* is *false* and *finished* is *false* then this thread must wait until the visiting iterator thread has reached the next element.

```

public synchronized boolean hasNext() {
    if (canceled)
        return false;
    checkValidity();
    while (!hasItem && !finished)
        try { wait(); } catch (InterruptedException ie) {}
    return hasItem;
}

```

The synchronized *next* method returns the next element in the iteration order, and advances the locator.

```

public synchronized E next() {
    if (!hasNext())                                //has next element only when hasNext is true
        throw new NoSuchElementException();
    hasItem = false;                                //about to consume nextItem
    notify();                                       //unblock the visiting thread upon return
    return nextItem;                                //return the next item
}

```

Since the visiting iterator merely traverses the elements of the collection as a visitor, it does not have access to the internal representation, as would typically be the case for a tracker or marker. Furthermore, it does not have the ability to change the traversal algorithm supplied by the collection. Therefore, the visiting iterator does not support the *get*, *inCollection*, *advance*, *retreat*, and *remove* methods. These methods all throw an *UnsupportedOperationException*.

```

public E get() { throw new UnsupportedOperationException(); }
public boolean inCollection() { throw new UnsupportedOperationException(); }
public boolean advance() { throw new UnsupportedOperationException(); }
public boolean retreat() { throw new UnsupportedOperationException(); }
public void remove() { throw new UnsupportedOperationException(); }

```

Any modifications to the data structure while the visiting iterator is in progress could cause exceptions in the data structure methods. Thus it is not safe to allow the user to continue when there have been concurrent modification exceptions. Therefore, both *ignoreConcurrentModifications* and *ignorePriorConcurrentModifications* also throw an *UnsupportedOperationException*.

```

public void ignoreConcurrentModifications(boolean ignore) {
    throw new UnsupportedOperationException();
}

public void ignorePriorConcurrentModifications() {
    throw new UnsupportedOperationException();
}

```

8.6 Performance Analysis

The *getCapacity*, *getComparator*, *getSize*, and *isEmpty* methods, clearly take constant time. The time complexity for the *compare* and *equivalent* methods depend on the *compare* method of the comparator *comp*, which we assume takes constant time. However, it is important to remember, that depending on the complexity of the objects being compared, there can be significant differences in the actual cost of the *compare* method.

The *clear*, *contains*, *toArray*, *toString*, *traverseForVisitor*, *accept*, and *writeElements* all use the iterator to traverse the data structure, which typically takes linear time. The performance analysis section in the chapter for each data structure will discuss the time complexity for iteration.

The static *getElementByRank* has expected linear time cost (see Section 11.8). The *c1.addAll(c2)* method calls *c1.add(o)* for each element $o \in c$. Thus the resulting collection has size $|c1| + |c2|$, so the time complexity for this *addAll* implementation is $O(f_a(|c1| + |c2|))$ where $f_a(m)$ is the time complexity for *c1.add* method when $|c1| = m$. Similarly, the *c1.retainAll(c2)* method calls *c1.contains(o)* for each element $o \in c$, and then possibly calls *c2.remove(o)*. Thus, the resulting time complexity for *retainAll(c)* is $O(f_c(|c1|) + f_r(|c2|))$, where f_c is the time complexity for *c1.contains*, and f_r is the time complexity for *c2.contains*.

Finally, the *AbstractLocator* constructor and its provided methods clearly take constant time.

8.7 Quick Method Reference

AbstractCollection Public Methods

- p. 98 void **accept**(Visitor<? super E> v)
- p. 100 void **addAll**(Collection<? extends E> c)
- p. 100 void **clear**()
- p. 97 boolean **contains**(E value)
- p. 99 void **ensureCapacity**(int capacity)
- p. 96 int **getCapacity**()
- p. 96 Comparator<? super E> **getComparator**()
- p. 98 E **getEquivalentElement** (E target)
- p. 96 int **getSize**()
- p. 96 boolean **isEmpty**()
- p. 100 void **retainAll**(Collection<E> c)
- p. 97 Object[] **toArray**()
- p. 97 E[] **toArray**(E[] array)
- p. 98 String **toString**()
- p. 99 void **trimToSize**()

AbstractCollection Internal Methods

- p. 97** `int compare(E e1, E e2)`
- p. 97** `boolean equivalent(E e1, E e2)`
- p. 99** `void traverseForVisitor(Visitor<? super E> v)`
- p. 98** `void writeElements(StringBuilder s)`

AbstractCollection.VisitingIterator Public Methods

- p. 103** `VisitingIterator()`
- p. 103** `void cancel()`
- p. 104** `E get()`
- p. 104** `boolean hasNext()`
- p. 101** `void ignoreConcurrentModifications(boolean ignore)`
- p. 101** `void ignorePriorConcurrentModifications()`
- p. 101** `E next()`
- p. 103** `void run()`
- p. 103** `void visit(E item)`

AbstractCollection.VisitingIterator Internal Methods

- p. 101** `void checkValidity()`
- p. 101** `void updateVersion()`

AbstractCollection.AbstractLocator Public Methods

- p. 101** `void ignoreConcurrentModifications(boolean ignore)`
- p. 101** `void ignorePriorConcurrentModifications()`
- p. 101** `E next()`

AbstractCollection.AbstractLocator Internal Methods

- p. 101** `void checkValidity()`
- p. 101** `void updateVersion()`

Chapter 9

Positional Collection ADT

package collection.positional

Collection<*E*>

↑ **PositionalCollection**<*E*>

Often an application needs to maintain a collection of elements that are accessed via their position in a line (with 0 being the position of the first element in the line) or via their location relative to other elements in the line. In a positional collection, the iteration order is given by: position 0, position 1, ..., position $n - 1$. Depending on the needs of the application program, either an array-based or list-based data structure may be best. We first describe how the *PositionalCollection* interface extends the *Collection* interface and then discuss the issues related to selecting the best data structure.

We have chosen to define the *PositionalCollection* interface so that after each call to the *add* method, all positions from 0, ..., $size' - 1$ are in use (possibly holding a *null* data value). An alternative approach would be to allow the user to add to any position $p \geq 0$ and with any unoccupied intermediate positions assigned *null*. This alternative approach would enable a user to create a sparse positional collection without calling *add* for each position that will be empty. However, the drawback of such an approach is that an erroneous application program could create a much larger positional collection than desired, and such an error might not be detected easily. If desired, a method *pad(newSize)* that increases the capacity of the underlying representation to *newSize*, padding the unused elements with *null*, could be introduced.

9.1 Interface

For our *PositionalCollection* ADT we present an interface that is similar to Java's *List*. The semantics of *add* from the *Collection* interface is changed to include the semantics that the new element is always added to the end of the collection. That is, *add* and *addLast* have the same semantics. We include *addLast* to be consistent with Java's interface. In addition, the following methods are those added to the methods inherited from the *Collection* interface. Some of these, notably those concerning ordering and sorting, do not appear in Java's *List* interface. We have omitted some methods provided in Java's *List* that can be easily implemented using the methods provided here. This choice enables us to focus on the methods that illustrate how the data structures work.

PositionalCollection(): Creates a new empty positional collection with a default initial capacity.

PositionalCollection(int capacity): Creates a new empty positional collection with the given capacity.

void add(int p, E value): Inserts *value* into position *p*. The position of the elements that were at positions $p, \dots, size - 1$ increase by one. It is required that $0 \leq p \leq size$ (i.e., the new element is to be put in an existing position or at the first unused position). Otherwise a *PositionOutOfBoundsException* is thrown.

void addFirst(*E value*): Inserts *value* at the front (position 0) of this collection. The positions of the existing elements are increased by one.

void addLast(*E value*): Inserts *value* at the end of this collection (position *size*).

***E* get(int *p*):** Returns the element at position *p*. It is required that $0 \leq p \leq \text{size}-1$ (i.e., that the position exists in the collection). Otherwise a *PositionOutOfBoundsException* is thrown.

PositionalCollectionLocator<*E*> iterator(): Returns a positional collection locator that has been initialized to FORE. It replaces the AbstractCollection *iterator* method to return the more specialized positional collection locator.

PositionalCollectionLocator<*E*> iteratorAtEnd(): Returns a locator that has been initialized to AFT. As with the *iterator* method, this method also enables navigation.

int positionOf(*E value*): Returns the position of the first occurrence (if any) of an element equivalent to *value*. It throws a *NoSuchElementException* if there is no equivalent element. (The Java collections return -1 when there is no element with the given value, but throwing an exception will better enable a problem to be detected if an application program did not consider the possibility of -1 being returned.)

***E* remove(int *p*):** Removes the element at position *p* and returns it. The positions of the elements originally at positions $p + 1, \dots, \text{size} - 1$ are decremented by 1. It is required that $0 \leq p \leq \text{size} - 1$ (i.e., that the given position exists in the current collection). Otherwise a *PositionOutOfBoundsException* is thrown.

***E* removeFirst():** Removes the element at the front (position 0) of this collection and returns it. The positions of all other elements are decremented by 1. If the collection is empty, a *NoSuchElementException* is thrown.

***E* removeLast():** Removes the element at the end (position $n-1$) of this collection and returns it. If the collection is empty, a *NoSuchElementException* is thrown.

void removeRange(int fromPosition, int toPosition): Removes the elements from position *fromPosition* to position *toPosition*, inclusive. The positions for elements $\text{toPosition} + 1, \dots, \text{size} - 1$ are decremented by $\text{toPosition} - \text{fromPosition} + 1$ (the number of elements removed). It is required that $0 \leq \text{fromPosition} \leq \text{toPosition} \leq \text{size} - 1$. A *PositionOutOfBoundsException* is thrown when either of the arguments is not a valid position, and an *IllegalArgumentException* is thrown when *fromPos* is greater than *toPos*.

***E* set(int *p*, *E value*):** Replaces the element at position *p* by the given *value*, and returns the element that had been in position *p*. It is required that $0 \leq p \leq \text{size}-1$. Otherwise, a *PositionOutOfBoundsException* is thrown.

void swap(int *a*, int *b*): Interchanges the element at position *a* with the element at position *b*. It is required that $0 \leq a \leq \text{size}-1$ and $0 \leq b \leq \text{size}-1$. Otherwise, a *PositionOutOfBoundsException* is thrown.

The following additional methods support ordering the elements of positional collection with a variety of algorithms.

void bucketSort(Bucketizer<? super *E*> bucketizer): Sorts this collection using bucket sort with the given bucketizer.

void heapSort(): Sorts this collection with heap sort using the default comparator.

void heapsort(Comparator<? super E> comp): Sorts this collection with heap sort using the provided comparator.

void insertionsort(): Sorts this collection with insertion sort using the default comparator.

void insertionsort(Comparator<? super E> comp): Sorts this collection with insertion sort using the default comparator.

void mergesort(): Sorts the positional collection with merge sort using the default comparator.

void mergesort(Comparator<? super E> comp): Sorts this collection with merge sort using the default comparator.

void quicksort(): Sorts this collection with quicksort using the default comparator.

void quicksort(Comparator<? super E> comp): Sorts this collection with quick sort using the default comparator.

void radixsort(Digitizer<? super E> digitizer): Sorts this collection using radix sort with the provided digitizer.

void treesort(): Sorts this collection with tree sort using the default comparator.

void treesort(Comparator<? super E> comp): Sorts this collection with tree sort using the provided comparator.

We also include methods to find a desired element according to its rank in the ordering.

E repositionElementByRank(int r): Modifies the positional collection so element in position r is in its proper sorted order when using the default comparator, and returns that element. This method does not completely sort the collection, but it may mutate it.

E repositionElementByRank(int r, Comparator<? super E> comp): Modifies the positional collection so element in position r is in its proper sorted order when using the provided comparator, and returns this element. This method does not completely sort the collection, but it may mutate it.

Critical Mutators for PositionalCollection: *add, addFirst, addAll, clear, heapsort, insertionsort, mergesort, quicksort, radixsort, remove, removeFirst, removeRange, repositionElementByRank, retainAll, swap, treesort*

9.2 Positional Collection Locator Interface

package collection.positional

Locator<E>

⌞ ***PositionalCollectionLocator<E>***

We extend the *Locator* interface by adding the following methods specific to a positional collection.

Locator<E> addAfter(E value): Inserts *value* immediately after the object referenced by this locator and returns a fresh locator at the position of insertion. If desired, a separate method could be written that does not return a locator.

int getCurrentPosition(): Returns the position within the collection for this locator.

Critical Mutators for PositionalCollectionLocator: *addAfter, remove*

9.3 Terminology

In addition to the definitions given in Chapter 7, we use the following definitions throughout our discussion of the PositionalCollection ADT and the data structures that implement it.

- We use u_p to denote the element in the p^{th} position of the collection. That is, the positional collection is denoted by $\langle u_0, u_1, \dots, u_{n-1} \rangle$.
- We say that a position is a **valid position** if it is between 0 and $n - 1$ (inclusive). We define a *PositionOutOfBoundsException(int position)* that extends *RuntimeException*. It is thrown when an invalid position is passed to a method that requires a valid position.
- For tracker t , if t tracks u_p for $p \in \{0, 1, \dots, \text{size}-1\}$, we say that t is **at position** p .
- Logically we treat FORE as position -1 and AFT as position *size*.
- When tracker t is tracking element o at position p for $p \in \{0, 1, \dots, n - 1\}$, and then element o is removed (by any mechanism), we say t is **between positions** $p - 1$ and p .

For a tracker t that is between positions $p - 1$ and p , the locator method call $t.\text{advance}()$ would move the tracker to position p and the locator method call $t.\text{retreat}()$ would move the tracker to position $p - 1$. For further discussion, see Section 5.8.

9.4 Competing ADTs

We briefly discuss ADTs that may be appropriate in a situation when a PositionalCollection is also being considered.

Mapping ADT: There are many situations when each element placed in a collection is given a unique identifier (created when the element is allocated). Sometimes, the identifiers will be $0, 1, 2, \dots$. While the application could treat the elements as if there were placed in a line in the order of the unique identifiers, the position does not need to be updated as changes are made, and the elements need not be ordered based on the position. Therefore, a positional collection is not the right choice. For this situation, the Mapping ADT is a better choice. More specifically, u_p is stored as a mapping entry $p \rightarrow u_p$. The Mapping ADT provides both constant time access via the key (p) and also constant-time insertion and removal of elements.

TaggedOrderedCollection ADT: As discussed further in the next section, all positional collection implementations either require linear time to locate an arbitrary element via its position

(as opposed to using a locator), or require linear time to add or remove an element at an arbitrary position. If this linear time cost is problematic for the application and the positions are just being used to enforce a relative order among the elements, where the absolute position is not important, then the TaggedOrderedCollection ADT should be considered, where the position is used as a tag for u_p . With such an implementation, it is possible to locate an element by position, to add an element at any position, or to remove an element from any position in logarithmic time. Such an implementation permits gaps in the tag sequence. Furthermore, floats, instead of integers may be used as tags to permit placement between any pair of successive elements.

Set ADT: Consider a Set ADT when the purpose of the collection is to maintain a subset of the elements $\{0, 1, \dots, k\}$. More specifically, one could imagine trying to use a positional collection to represent a set by associating a position index with each possible element of the universe, and letting u_p be *null* exactly when the element associated with p is not a member of the set. However, a Set ADT would provide a more natural solution and offers more space efficient implementations.

Buffer ADT: The Buffer ADT is a special case of the PositionalCollection in which all accesses and modifications (except those made via a locator) are made at the front or end of the collection. If efficient access (except through a locator) is only required at the ends, then a buffer is a better choice than a positional collection.

Queue ADT: The Queue ADT is an even more specialized case of the Buffer. If the line is being used as a first-in-first-out line then, a queue is the right choice.

Stack ADT: The Stack ADT is also a specialized case of the Buffer. If the line is being used as a last-in-first-out line then, a stack is the right choice.

9.5 Selecting a Data Structure

After deciding that a positional collection is the best ADT for an application, the next design decision is whether to use an array-based or a list-based implementation. To aid in the process of selecting the best PositionalCollection data structure, consider which of the following properties are most important for the desired application. The properties are listed from the most to least significant in terms of how much they should affect the decision process. Table 9.1 provides a visual summary of these trade-offs.

Access by position. The primary advantage of an array-based approach is the ability to access position p in constant time regardless of the value of p . A drawback of a list-based approach is that a brute-force linear search is required to locate the element at position p . Even when using a doubly linked list in which the search can progress from the front or back, the time required to access the p^{th} element is $O(\min(p + 1, n - p))$. Thus, if the desired application needs constant time access by position then an array-based approach should be used unless one of the drawbacks of an array is unacceptable.

Efficiency of adding or removing elements. The primary advantage of a list-based approach is that once the desired position has been located, an element can be inserted in that position in constant time. Furthermore, with a doubly linked list, an element can also be removed in constant time. For an array-based approach, in order to insert an element into the array, all of the elements in later positions must be moved forward in the array, which takes time

Key ⦿ Excellent ○ Good ● Fair • Not required							
	Array	Circular Array	DynamicArray	DynamicCircular Array	TrackedArray	SinglyLinkedList	DoublyLinkedList
access by position near middle	⦿	⦿	⦿	⦿	⦿	○	○
adding to middle portion (once located)	○	○	○	○	○	⦿	⦿
removing from middle portion (once located)	○	○	○	○	○	○	⦿
access to front	⦿	⦿	⦿	⦿	⦿	⦿	⦿
adding to front	●	⦿	●	⦿	⦿	⦿	⦿
removing at front	●	⦿	●	⦿	⦿	⦿	⦿
access to back	⦿	⦿	⦿	⦿	⦿	⦿	⦿
adding to back	⦿	⦿	⦿	⦿	⦿	⦿	⦿
removing from back	⦿	⦿	⦿	⦿	⦿	●	⦿
determining the position of a locator	⦿	⦿	⦿	⦿	⦿	●	○
support of a tracker					✓	✓	✓
space usage	⦿	⦿	⦿	⦿	○	○	●
ease in adjusting the capacity	●	●	●	●	●	⦿	⦿
automatic resizing			✓	✓	✓	•	•

Table 9.1 A summary of the broad trade-offs between list-based and array-based data structures for implementing the PositionalCollection ADT.

proportional to the number of elements being moved. A circular array partly rectifies this problem allowing efficient insertion to the front portion of the array. However, even when using a circular array, it still takes linear time to add an element to the middle portion.

It is important to keep in mind that the real advantage of a list-based approach for adding or removing elements in the middle portion of the collection is only realized if there is an efficient way to locate the desired position. In general, this is achieved either through a tracker associated with a given element or a marker that is being used to iterate within the collection.

Determining the position for a locator. An advantage of an array-based approach is that the *Locator* *getPosition* method takes constant time. In contrast, for a list-based approach this method must follow $p + 1$ references with a singly linked list, or $\min(p + 1, n - p)$ references with a doubly linked list to determine the position of the locator. While it might appear that this cost could be reduced by including an instance variable *position* to each tracker to record its position, whenever an element is added or removed, this *position* would need to be updated for all trackers whose corresponding elements were moved, destroying a primary advantage of using a list-based approach. So, if *getPosition* must run in constant time, then an array-based approach is a better choice.

Ease in supporting a tracker. Another advantage of a list-based approach is that there is minimal overhead associated with maintaining a tracker since in any pointer-based data structure the elements are not moved within memory but instead references are modified. While the asymptotic time complexity for array-based data structures is the same whether a marker or tracker is supported, there is some overhead in terms of memory usage and computation cost required to support a tracker.

Space usage. It may seem that since a list-based implementation is elastic (always holding exactly n nodes), that it makes the most efficient use of space. However, consider a situation in which the number elements to be placed in the collection is known when the collection is allocated, and further that there is no need for a tracker. For example, the application may be designed to run on a portable device that has very limited memory. In this situation an array that is allocated to exactly the right size provides the most efficient space usage. In particular, if there are n elements stored, then the array only uses n references to the elements along with a few instance variables such as the size and capacity of the array. In fact, even a dynamic array on average uses only about $1.5n$ references. In contrast, a singly linked list without support for a tracker would require $2n$ references, a reference to the element and reference to the next list node for each element. Thus, an array-based approach is more space efficient than a list-based approach. The primary limitation of any array-based approach is that elements cannot be added or removed efficiently from the middle portion of the collection.

Ease in adjusting the capacity. Another advantage of a list-based data structure is that new elements can be allocated as needed and removed elements can be reclaimed through the garbage collector. Thus with a list-based approach, the internal representation always contains just one node per element. In contrast, since an array is a contiguous block of memory, it must be allocated with a given capacity. To enlarge or reduce the size of an array has linear cost, since a new array with the desired capacity must be allocated, and all existing elements must be copied to this new array. Thus, it is too expensive to resize the array whenever the size of the collection changes. Unless the application program knows how many elements are going to be held by a positional collection at the time when the constructor is called, the only practical option is to occasionally resize the array when there is no spare capacity or whenever it is greatly underutilized. While the amortized cost for these resizing methods is constant, the actual cost when the array is resized is linear. Thus, if it is expected that the number of elements

held in the collection will change dramatically over time, then this is an indication to use a list-based approach. This issue is secondary to those discussed earlier since the additional amortized cost for resizing an array is relatively small.

Sorting the collection. For some applications there is a need to frequently sort the collection when there are multiple ways of ordering the elements. For example, in managing a collection of mail messages the user may want to sort the elements by date, and then later sort them by sender. The ability to access any position in constant time makes array-based representations slightly faster, in general, for sorting.

Working with fragmented memory. Since an array is a contiguous block of storage, if the memory is fragmented with no block large enough for the needed array then either a run-time exception will occur or time must be spent restructuring memory which would be very costly. Hence, in the somewhat uncommon situation in which the memory may become overly fragmented, a list-based approach is best.

The above discussion should provide the needed guidance to choose between an array-based or list-based implementation. For example, if an application must locate a desired element, then an array-based implementation should be used. However if elements will be inserted and removed in the middle of the collection via a locator, then a list-based approach should be used. If it is important to both efficiently locate any element by its position and to add or remove elements near the middle of the collection, then the Positional Collection ADT may not be the right choice. See the discussion on the TaggedOrderedCollection ADT in [Section 9.4](#).

Next, assuming that one has decided to use a positional collection and has made a choice between an array-based and a list-based implementation, we separately discuss alternative data structures that use an array-based approach and a list-based approach. We close this section with a summary that compares the positional collection data structures.

9.5.1 Tradeoffs among Array-Based Data Structures

The built-in Java array provides a foundation for implementing a positional collection where array slot p holds the element in position p . However, the primitive Java array has the limitation that it cannot be resized once allocated. The array-based data structures for the positional collection can be viewed as extensions of the Java primitive array in which the array can be resized as needed (either under the application program control via a method call or by an automatic resizing protocol). Another drawback of directly using a Java primitive array as the underlying representation is that to add (or remove) an element at index i , all elements currently at indices $i, \dots, n - 1$ must be shifted. For example, if a new element is inserted at position 0 then all elements currently in the collection must be copied one slot to the right which will require linear time.

These two drawbacks of the Java primitive array leads to the following two orthogonal choices to consider: whether to use a **dynamic array** that automatically resizes, and whether to use a **circular array** which enables efficient insertion/deletion at the front of the array. Note that a dynamic array is not the same thing as a **dynamically-allocated array**, which is a standard fixed-size array whose size is selected at runtime.

Dynamic array or non-dynamic array. When using a **non-dynamic array**, the application program is responsible for managing the capacity of the underlying Java primitive array via the *ensureCapacity* and *trimToSize* methods. When using a non-dynamic array, if an attempt is made to add an element when the underlying array is full, an *AtCapacityException* is thrown that the application program can catch and use as an indication the capacity needs to be increased. If done properly, then space usage is minimized. However, since resizing the array

takes linear time, calling *ensureCapacity* or *trimToSize* too frequently will lead to poor performance. In addition, the application program can waste memory by providing an argument to *ensureCapacity* that is much larger than needed.

When using a **dynamic array**, support is provided to automatically resize the underlying array as needed. In Section B.5, we prove that by doubling the capacity of the array when it is full, and halving the capacity of the array when only 1/4 of the slots are in use, there is a constant amortized cost for both inserting and removing elements from the array. Intuitively, the cost of the resizing is constant when divided among all of the add and remove operations that caused the resizing to be needed. Thus, the resizing cost does not change the asymptotic complexity of any operations to be performed on the array. While the worst-case space usage is $4n$ (which occurs when a significant number of elements have recently been removed from the collection), more typically the underlying array capacity is much smaller. If no deletions occur then the worst case space usage is $2n$ and this only occurs when the array has just been doubled and no new insertions have yet been made.

A natural question to ask is: why not always use a dynamic array? While the dynamic array can do everything that a non-dynamic array can, there is a cost associated with each insertion and deletion in checking to see if the array needs to be resized. Thus, if the application program knows approximately how many elements will be stored in the positional collection then the non-dynamic array implementation will be more efficient.

Non-circular array or circular array. Recall that one drawback of the Java primitive array is that it takes $O(n - p)$ time to add an element at position p since the elements in positions $p, \dots, n - 1$ must be copied to the next array slot. A **circular array** is designed to partly rectify this problem by adjusting the underlying array index where the collection begins. In particular, the element at position 0 can be held in the Java primitive array at any index between 0 and $n - 1$. For $p \in \{1, \dots, n - 1\}$, the element at position p is held in the array slot immediately after that holding the element at position $p - 1$ where the Java array is viewed as wrapping (to form a circle) with position 0 immediately after position $n - 1$. Hence the name circular array. See Figure 12.1 on page 172 for an illustration of a circular array. When using a circular array, the time complexity of adding or removing an element at position p is $O(\min(p + 1, n - p))$ since either all elements in positions $0, \dots, p$ or all elements in positions $p, \dots, n - 1$ can be moved.

If the desired application will insert or remove elements near the front of the positional collection, then a circular array should be used. The primary cost for a circular array is the computation required to determine which index in the underlying array holds the element at position p . This cost is small, so a circular array is best unless the application program very rarely adds or removes elements in the front half of the collection.

There is one last decision that must be made – Does the application require a tracker for each element inserted in the collection? When a tracker must be supported, some additional structure is needed so that the tracker can be modified as elements are moved. While one could view this option as a third orthogonal feature, for ease of exposition, we just present one tracked array-based structure which is built upon a dynamic circular array. The tracked array data structure is implemented as a wrapper, so any of the other array-based data structures could instead be used. However, there is some space and time cost associated with supporting a tracker.

9.5.2 Tradeoffs among List-Based Data Structures

The only option to decide once a list-based approach has been selected is whether to use a **singly linked list** (with only a reference to the element at the next position) or a **doubly linked list** (with references to the elements at both the next and previous positions). The advantage of the singly

linked list is that it uses less memory since the previous references are not needed. Also, a small amount of computation is saved in a singly linked list since there are only half as many references to maintain. This leads to the following trade-offs:

Access by position. In a singly linked list it takes $O(p+1)$ time to access the element at position p , whereas in a doubly linked list it takes $O(\min(p+1, n-p))$ to access the element at position p . The one exception is that both implementations maintain a reference to the last element (position $n-1$) in the collection, so it can be located in constant time.

Removing an element via a tracker. In a singly linked list, even if an element to be removed has been located (perhaps using a tracker), it cannot be removed in constant time since the element before it in the list must be located which takes $O(p)$ time if the element to be removed is at position p . For a doubly linked list, once an element has been located it can be removed in constant time.

Locator retreat method efficiency. In a singly linked list it takes worst-case linear time to perform the *Locator retreat* method, whereas it takes constant time with a doubly linked list.

A singly linked list is often preferred to a doubly linked list because it reduces the space complexity. However, as discussed above, the lack of a reference to the previous list node makes the locator *remove* method inefficient. Thus, two additional methods are provided specifically for the `SinglyLinkedList` tracker to improve performance of removing items during iteration. Because there is no previous reference, these methods allow the tracker to look one element ahead in order to possibly remove the next item while the tracker still retains a reference to the list node that precedes it. The `getNextElement` method returns the element that is in the position that follows the locator, and the `removeNextElement` method removes the element that is in the position that follows the locator. Both of these methods run in constant time. In summary, the singly linked list should only be used if access by position is only needed for the front portion of the collection or for the last element, and if it is very uncommon for the application to call the locator *retreat* method.

9.6 Summary of Positional Collection Data Structures

Table 9.2 provides a high-level summary of the trade-offs between the data structures we present for the `PositionalCollection` ADT. This table can be used to help select the most appropriate data structure for a given application. In the context of Table 9.2, we define excellent performance as constant time, fair performance as $O(\min(p+1, n-p))$, and poor performance as $O(p)$ time. For the row “adjusts capacity as needed” we define excellent performance as worst-case constant time to adjust the capacity and fair performance as constant amortized time (i.e., if m operations occurred since the last resizing, it would take $O(m)$ time).

Array: The simplest of the positional collections, `Array` provides space for a fixed number of elements, which are stored in an underlying Java primitive array. Methods are provided for changing the capacity of the collection, but there is no support for automatic resizing. If the approximate size of the collection is known a priori then this is a very good choice for minimizing space usage. However, it can only efficiently support insertions and deletions near the end of the collection. This is an untracked implementation.

CircularArray: This array-based data structure allows element 0 of the positional collection to be in any slot of the underlying array, with the range of underlying indices wrapping around as needed. Managing this introduces a small amount of overhead, but enables efficient insertion

Key								
⊙	Excellent							
⊖	Very Good							
⊕	Marginal							
●	Fair							
•	Method does nothing							
Method		Array	Circular Array	Dynamic Array	DynamicCircular Array	Tracked Array	SinglyLinkedList	DoublyLinkedList
Positional Collection Methods	add(o), addTracked(o)	⊙	⊙	⊙	⊙	⊙	⊙	⊙
	add(pos,o), remove(pos)	●	⊖	●	⊖	⊖	●	⊖
	addFirst(o)	●	⊙	●	⊙	⊙	⊙	⊙
	addLast(o)	⊙	⊙	⊙	⊙	⊙	⊙	⊙
	addAll(C), per element in C	⊙	⊙	⊙	⊙	⊙	⊙	⊙
	clear(), per element	⊙	⊙	⊙	⊙	⊙	⊙	⊙
	contains(o)	●	●	●	●	●	●	●
	ensureCapacity(x), trimToSize()	●	●	●	●	●	•	•
	get(pos)	⊙	⊙	⊙	⊙	⊙	●	⊕
	iterator(), iteratorAtEnd()	⊙	⊙	⊙	⊙	⊙	⊙	⊙
	iteratorAt(o)	●	●	●	●	●	●	●
	positionOf(o)	●	●	●	●	●	●	●
	remove(o), removeRange(i,j)	●	●	●	●	●	●	●
	removeFirst()	●	⊙	●	⊙	⊙	⊙	⊙
	removeLast()	⊙	⊙	⊙	⊙	⊙	●	⊙
Other Issues	retainAll(C) per element in C	●	●	●	●	●	●	●
	set(pos,o), swap(pos1, pos2)	⊙	⊙	⊙	⊙	⊙	●	⊕
	accept(v), toArray(), toString(), per element	⊙	⊙	⊙	⊙	⊙	⊙	⊙
	typical space	⊙	⊙	⊖	⊖	●	⊕	●
Locator Methods	implements Tracked					✓	✓	✓
	automatic resizing		✓		✓	✓	•	•
	addAfter(o)	●	●	●	●	●	⊙	⊙
	advance(), get(), getNextElement()	⊙	⊙	⊙	⊙	⊙	⊙	⊙
	getPos()	⊙	⊙	⊙	⊙	⊙	●	⊕
	moveTo(loc), next()	⊙	⊙	⊙	⊙	⊙	⊙	⊙
	remove()	●	●	●	●	●	●	⊙
	removeNextElement()	●	●	●	●	●	⊙	⊙
	retreat()	⊙	⊙	⊙	⊙	⊙	●	⊙
	set(o)	⊙	⊙	⊙	⊙	⊙	⊙	⊙
		⊙	⊙	⊙	⊙	⊙	⊙	⊙

Time Complexity
⊙ O(1) time
⊖ O(min(p+1,n-p)) time
● O(n) time
• method does nothing

Space Usage
⊙ n cells
⊖ 1.5n cells, on average
⊕ 2n cells
● 4n cells

Table 9.2 Trade-offs among the data structures that implement the PositionalCollection ADT. For the array and circular array, it is assumed for the purposes of comparison that the application program knows exactly how many elements are going to be placed into the collection, and that the underlying array has size n .

and deletion near either the front or end of the collection. Like Array, there is no support for automatic resizing. This is an untracked implementation.

DynamicArray: This array-based data structure provides space for a fixed number of elements, which are stored in an underlying Java primitive array. However unlike Array and CircularArray, it performs automatic resizing. When the size of the collection is unknown or changing, the support for automatic resizing is important. However, the resizing introduces a small amount of additional computation whenever an element is added or removed, and the resizing method has linear cost (though the amortized cost is constant). On average, the size of the array is roughly 50% larger than that needed. As with Array and CircularArray, DynamicArray supports user-controlled resizing through *ensureCapacity* and *trimToSize*. When the size of the collection matches that of the user-specified capacity, the performance of a dynamic array is almost identical to that of an array. The advantage of using a dynamic array over an array is that when the initial estimate for the size is wrong, then the data structure automatically resizes, versus leaving it to the application program. This is an untracked implementation. The Java **ArrayList** data structure is most similar to this data structure.

DynamicCircularArray: This array-based data structure allows element 0 of the positional collection to be in any slot of the underlying array, with the range of underlying indices wrapping around as needed, and also performs automatic resizing. It is best if the size of the collection is not known a priori and if some insertions and/or deletions tend to occur near the front portion of the collection. As with DynamicArray, the application program can provide an initial size in the constructor and use *ensureCapacity* and *trimToSize* to optimize performance. From Table 9.2, it may appear that a dynamic circular array is superior to a dynamic array. In terms of asymptotic complexity, this is true. However, DynamicCircularArray one drawback. The *set* and *get* methods are slightly more expensive since the underlying index for the element at position p must be computed. While this is a very small cost, if an application never adds or removes from front of the collection, it is not worth this cost. This is an untracked implementation.

TrackedArray: This array-based data structure can wrap any of the other array-based data structures to create a tracked implementation of the wrapped data structure. In the provided implementation, TrackedArray wraps DynamicCircularArray. However, the provided implementation could easily be changed to use any of the other array-based data structures. Since the maintenance of the trackers requires some additional structure as compared to that needed for a marker, a tracked array should be used only if it is important to be able to track elements. Observe that the tracker does not reduce the cost of shifting array elements when an element is added or removed, even through a tracker. The tracker would be useful when the application needs to determine the position of a tracked element in constant time.

SinglyLinkedList: The simplest of the list-based positional collections, SinglyLinkedList maintains a linked list where each list node only references the next element in the list. Our implementation also maintains a reference to the last item in the list. It is a tracked implementation. When using a locator to iterate through the collection, in constant time a new element can be added or the next element in the collection can be removed. The primary drawback of using a singly linked list is that it takes $O(p)$ time to locate the element at position p for $0 \leq p \leq n-2$. Another weakness of SinglyLinkedList is that it takes $O(p)$ time to find the element that precedes the element at position p . Hence, the *retreat* method cannot be efficiently implemented. Moreover, in order to efficiently remove a tracked element from position p of the collection, it takes $O(p)$ time.

DoublyLinkedList: This list-based data structure extends SinglyLinkedList to have each list item also maintain a reference to the previous list node. The primary advantage of a doubly

linked list, over all of the other data structures, is that through a locator an element can be added or removed from the middle portion of the list in constant time. Also, it takes $O(\min(p + 1, n - p))$ time, instead of $O(p)$ time for `SinglyLinkedList` to access the element at position p without the use of a locator. Furthermore, the *retreat* method takes constant time. The primary disadvantage of `DoublyLinkedList` is that including a previous reference adds space overhead and all methods that modify the structure of the list must adjust twice as many references.

Figure 9.3 shows the class hierarchy for the data structures presented in this book for the `PositionalCollection` ADT.

9.7 Further Reading

Brodnik et al. [32] present an implementation for a resizable array that has constant amortized cost to add or remove elements from the array and guarantees an additive $O(\sqrt{n})$ upper bound on the additional space used. They achieve this goal by using a representation where the array is composed of a set of arrays with varying sizes. In contrast, the dynamic array presented here uses up to four times as much space as needed, but with less overhead for getting elements by position.

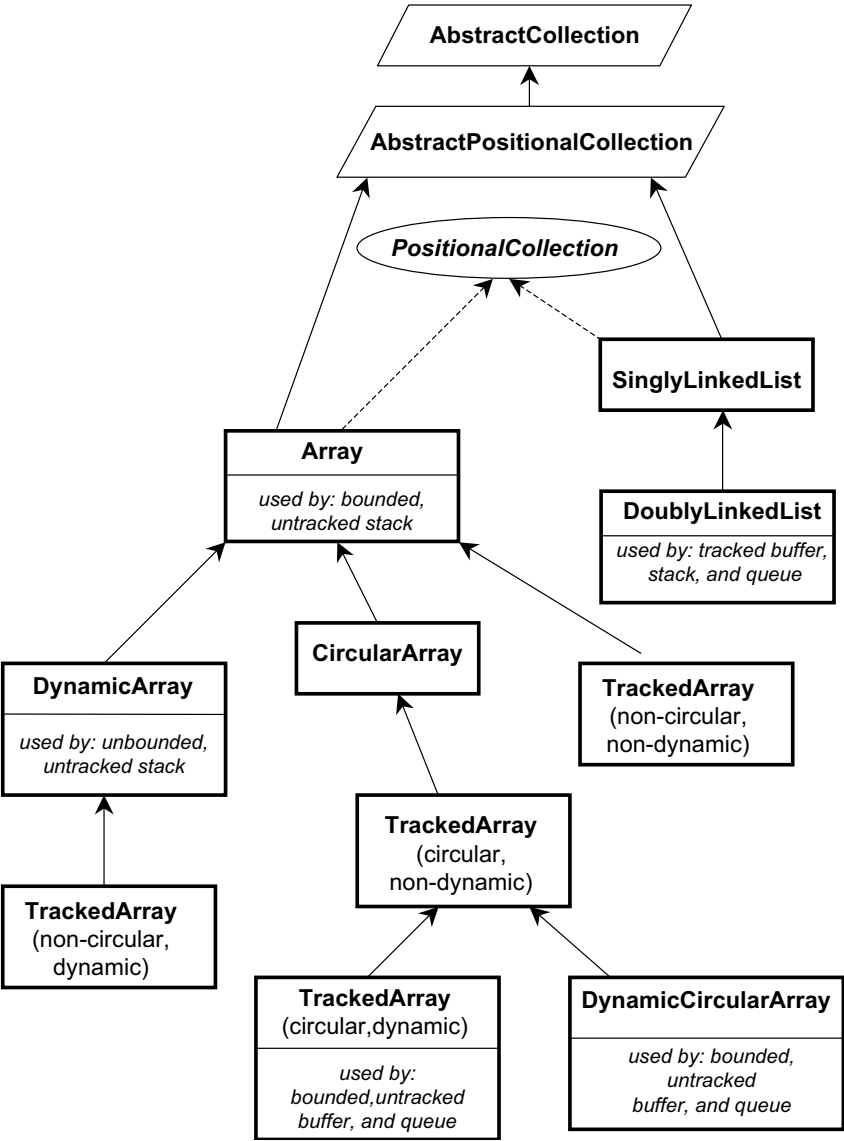


Figure 9.3
The class hierarchy for the positional collection data structures. Abstract classes are shown as parallelograms, concrete classes as rectangles, and interfaces as rounded rectangles. Solid lines represent subclass relationships, with parents above children on the page. A dashed line from a class to an interface indicates that the class implements that interface.

Chapter 10

Abstract Positional Collection

10.1 Abstract Positional Collection

package collection.positional

AbstractCollection<E> implements *Collection*<E>

└─ **AbstractPositionalCollection**<E> implements *PositionalCollection*<E>

10.2 Internal Representation

The `AbstractPositionalCollection` provides a basis for defining concrete positional collections by defining a useful set of methods that can be implemented for any positional collection in terms of the public methods from the *PositionalCollection* interface. The internal representation can be any positional collection data structure.

The constructor that takes no parameters creates an empty positional collection that uses the default comparator.

```
public AbstractPositionalCollection() {
    this(Objects.DEFAULT_EQUIVALENCE_TESTER);
}
```

The constructor that takes *comp*, the function used to compare two elements, creates an empty positional collection that uses the given comparator.

```
public AbstractPositionalCollection(Comparator<? super E> comp) {
    super(comp);
}
```

The *toString* method is modified so that it returns a comma-separated string showing the elements in the iteration order. Angle brackets mark the beginning and the end of the collection.

```
public String toString() {
    StringBuilder s = new StringBuilder("<");
    writeElements(s);
    s.append(">");
    return s.toString();
}
```

Correctness Highlights: Follows from that of the *writeElements* method from the *AbstractCollection* class. ■

The *addFirst* method that takes *value*, the element to insert, inserts it at the front (position 0) of the collection.

```
public void addFirst(E value) {
    add(0, value);
}
```

The *addLast* method that takes *value*, the element to insert, inserts it at the end (position *size*) of the collection.

```
public void addLast(E value) {
    add(value);
}
```

All methods that return an iterator are overridden to instead return a positional collection locator for more specific use of the return value without the need to cast. The abstract *iterator* method creates a new positional collection locator that starts at FORE.

```
public abstract PositionalCollectionLocator<E> iterator();
```

The abstract *iteratorAt* method takes *pos*, the user position of an element, and returns a positional collection locator that is at the given position. It throws a *NoSuchElementException* when the given position is not a valid position.

```
public abstract PositionalCollectionLocator<E> iteratorAt(int pos);
```

The abstract *iteratorAtEnd* method creates a new positional collection locator that starts at AFT.

```
public abstract PositionalCollectionLocator<E> iteratorAtEnd();
```

10.3 Quick Method Reference

AbstractPositionalCollection Public Methods

- p. 121 *AbstractPositionalCollection*()
- p. 121 *AbstractPositionalCollection*(*Comparator*<? super E> comp)
- p. 98 void **accept**(*Visitor*<? super E> v)
- p. 100 void **addAll**(*Collection*<? extends E> c)
- p. 122 void **addFirst**(E value)
- p. 122 void **addLast**(E value)
- p. 100 void **clear**()
- p. 97 boolean **contains**(E value)
- p. 99 void **ensureCapacity**(int capacity)
- p. 96 int **getCapacity**()
- p. 96 *Comparator*<? super E> **getComparator**()
- p. 98 E **getEquivalentElement** (E target)

- p. 96 int **getSize()**
- p. 96 boolean **isEmpty()**
- p. 100 void **retainAll**(Collection<E> c)
- p. 97 Object[] **toArray()**
- p. 97 E[] **toArray**(E[] array)
- p. 98 String **toString()**
- p. 99 void **trimToSize()**

AbstractPositionalCollection Internal Methods

- p. 97 int **compare**(E e1, E e2)
- p. 97 boolean **equivalent**(E e1, E e2)
- p. 99 void **traverseForVisitor**(Visitor<? super E> v)
- p. 98 void **writeElements**(StringBuilder s)

Chapter 11

Array Data Structure

package collection.positional

AbstractCollection<E> implements *Collection*<E>

└ AbstractPositionalCollection<E> implements *PositionalCollection*<E>

└ Array<E> implements *PositionalCollection*<E>

Uses: Java primitive array

Used By: AbstractCollection (Chapter 8), Stack (Chapter 19), SortedArray (Chapter 30), BTree (Chapter 36), DirectAddressing (Chapter 21), and TaggedCollectionWrapper (Section 49.6).

Strengths: Random access by position is supported in constant time. The array can be resized by the user if needed. In constant time an element can be added to or removed from the back of the collection.

Weaknesses: Inserting or removing from the front of the array has linear cost. The user is responsible for calling a method to ensure sufficient capacity at appropriate times. If resizing is not done at appropriate times, one of the following problems can occur:

- A runtime exception will occur if there is an attempt to insert an element into a collection that is at capacity.
- If the array capacity is significantly larger than needed, space is wasted.
- Since resizing the array takes linear time, calling *ensureCapacity* or *trimToSize* too frequently will lead to poor performance. As discussed in Chapter 13, the dynamic array data structure ensures amortized constant cost for the resizing.

Critical Mutators: *add* (except when $p = \text{size}$), *addFirst*, *addAll*, *clear*, *heapsort*, *insertionsort*, *mergesort*, *quicksort*, *radixsort*, *remove*, *removeFirst*, *removeRange* (except when $\text{toPosition} = \text{size} - 1$), *repositionElementByRank*, *retainAll*, *swap*, *treesort*

Competing Data Structures: While the Array class is built upon Java's built-in array, it does introduce some overhead when accessing the array. If the application only needs *get* and *set*, and the required array size is known at allocation time, it would be more efficient to directly use Java's built-in array.

A dynamic array (Section 13.1) is preferred when automatic resizing is desirable. If the application frequently adds or removes elements near the front and back of the collection, a circular array (Chapter 12) or a dynamic circular array (Section 13.6) is best.

A list-based approach should be considered if elements are going to be inserted or removed frequently via a tracker or marker. The *addAfter*, *removeAfter*, and *remove* methods take linear time for any array-based approach if the marker or tracker is near the middle portion of the array. In

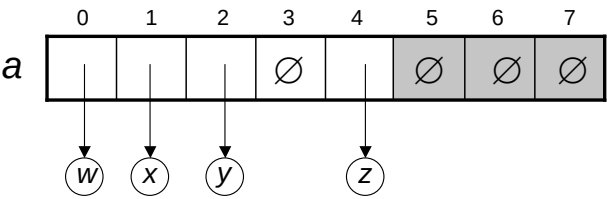


Figure 11.1
A populated example for an array instance containing the positional collection $\langle w, x, y, \text{null}, z \rangle$. Shaded slots are not in use.

contrast, for a doubly linked list (Chapter 16) all three of these methods take constant time. For a singly linked list (Chapter 15), *addAfter* and *removeAfter* take constant time, and only *remove* requires linear time.

11.1 Internal Representation

Elements are stored in an underlying Java primitive array. If the underlying array is full, an exception is thrown if there is an attempt to add an element to the collection. Recall that we use n to denote the number of elements in the collection.

Instance Variables and Constants: In addition to *size*, *DEFAULT_CAPACITY*, *NOT_FOUND*, *FORE*, *version*, and *comp* that are inherited from *AbstractCollection*, the *Array* class declares the underlying array *a*. The type of *a* is an array of objects (*Object[] a*) because Java does not support creating an array with a generic type, since type parameters are erased by the compiler.

`Object[] a;` *//the underlying array*

The capacity of the collection is *a.length* where $a.length \geq n$.

Populated Example: Figure 11.1 shows the Java underlying primitive array *a* used by the internal representation for an array containing the positional collection $\langle w, x, y, \text{null}, z \rangle$. An arrow is used to denote a non-*null* pointer to an element that can be of any type. The empty set ($\emptyset$) symbol denotes a *null* pointer, and the portion of *a* not in use is shaded gray. For the array illustrated in Figure 11.1, *size* is 5 and the capacity of *a* is 8. As illustrated by position 3 in this collection, it is permissible for an element of the collection to be *null*.

Abstraction Function: Let *a* be an *Array* instance. The abstraction function

$$AF(a) = \langle u_0, u_1, \dots, u_{size-1} \rangle \text{ such that } u_p = a[p].$$

Terminology: We use the following definitions throughout our discussion of the array-based structures that implement a *PositionalCollection*.

- Slots $0, 1, \dots, size-1$ of the underlying array are **in use**. All other slots are said to be **not in use**. Slots $size, \dots, a.length - 1$ in an array are not in use.
- Position p is a **valid position** if and only if $0 \leq p < size$.

Design Notes: The Array class illustrates several aspects of design that appear throughout this book.

- We decouple the method to compare elements for equivalence from the data structure implementations by allowing an application program to either use a default equivalence tester or provide an equivalence tester to the constructor.
- We use the **strategy design pattern** to decouple the mechanism to define an order of the elements (e.g., comparator, digitizer, or bucketizer) from the sorting algorithms and the selection algorithm that rely on this ordering.
- We use the **template method design pattern** in which the skeleton of an algorithm is written with some steps delegated to other methods enabling just those steps to be refined by overriding them in subclasses. The most significant use of this design pattern is in the methods that translate between the user position and the underlying index. By separating these methods from the rest of the computation used in many methods, we can apply them to a circular array by just replacing the methods that perform this translation between user position and the underlying index.

Optimizations: We have made some design choices that improve the extensibility of our implementations, but they do introduce a small amount of overhead. In particular, the representation accessors, *getIndex*, *getCurrentPosition*, *nextIndex*, *prevIndex*, *read*, and *write* are included so that the array can easily be extended to a circular array. However, the method call overhead could be removed by using the index and user position interchangeably (with no conversion method needed at all). Also, *nextIndex* and *prevIndex* could be made final, which would enable the Java compiler to in-line them thus removing the method call. However, these optimizations would need to be made in an alternate Array class that is not extended by CircularArray.

To reduce the overhead created by the translation between the user position and the underlying array index, we have chosen to have a few methods (namely, *closeGap* and *createGap*) directly use the position as the underlying index to increase efficiency. We made this decision since using *system.arraycopy* to move all elements in a contiguous (non-wrapped) portion of the underlying array is significantly more efficient than individually moving the elements. (The asymptotic time complexity is not changed.) CircularArray must override these methods.

To avoid the need to repeatedly check if a user-provided position is a valid position, this check is made only by the public methods. All internal methods that take a position as a parameter require that the position be valid. While this design decision reduces unnecessary overhead, it leaves open the possibility that an overriding method violates the requirement understood. If a small increase in time overhead is acceptable, the internal methods could check that the stated requirements are satisfied, and throw an exception when they are not.

Similarly, executing the internal method that swaps two elements necessitates the need to invalidate all locators, since continued iteration could cause an element to be seen twice or not at all. However, some algorithms, such as quicksort, call swap many times, and it would be inefficient to update the modification count for each swap. Thus, we have made a design decision to only invalidate the locators from the public methods, or an internal method that is called only once per call of the public method.

The merge sort algorithm when implemented on an array-based representation requires an auxiliary array for the merge. The most natural approach for providing this array would be to allocate an array for each call made to merge sort. However, such an approach would cause unnecessary overhead in both the allocation of an auxiliary array for each recursive call, and also in the resulting increase in garbage collection that it would create. We avoid this overhead by allocating an auxiliary array at the top-level call to merge sort, and then we pass this auxiliary array to the recursive merge

sort implementation. As discussed in Section 11.4.3, an in-place implementation of heap sort could be implemented if desired.

The *retainAll* method from the *AbstractCollection* class could be overridden to run in linear time by first marking all the elements to retain in an auxiliary array, and then in a single pass shifting them into the correct position. We have chosen to not include this optimization in our provided implementation.

11.2 Representation Properties

To simplify our discussion of correctness, we inherit *SIZE* and introduce three additional representation properties.

CAPACITY: The capacity of the underlying array must be at least as large as the number of elements held. That is, $a.length \geq size$.

PLACEMENT: The element in position p , in the user view, is the element placed at index p of the underlying array. More formally, $a[p] = u_p$ for all $p \in \{0, \dots, size-1\}$.

NONRETENTION: The data structure does not hold references to objects that are no longer elements of the collection. That is, $a[i] = null$ for all slots i that are not in use.

NONRETENTION prevents the array from unnecessarily retaining references to objects that might otherwise be eligible for garbage collection.

11.3 Methods

In this section we present the internal and public methods for the *Array* class.

11.3.1 Constructors

The most general constructor takes *capacity*, the desired initial capacity for the underlying array, and *equivalenceTester*, a user-provided equivalence tester, creates an array with the given capacity that uses the provided equivalence tester. It throws an *IllegalArgumentException* when *capacity* < 0.

```
public Array(int capacity, Comparator<? super E> equivalenceTester){
    super(equivalenceTester);
    resizeArray(capacity);
    size = 0;
}
```

Correctness Highlights: *SIZE* is satisfied since $size=n=0$. *CAPACITY* holds since *resizeArray* ensures that $a.length \geq 0$. *PLACEMENT* holds vacuously since $n = 0$. Finally, Java initializes all slots of an array to *null*, so *NONRETENTION* holds after *resizeArray*. ■

Several additional convenience constructors are provided to replace some parameters by the default values. Their correctness follows from that of the above constructor. The constructor with no arguments creates an array with a default initial capacity that uses the default equivalence tester.

```
public Array(){
    this(DEFAULT_CAPACITY, Objects.DEFAULT_EQUIVALENCE_TESTER);
}
```

The constructor that takes *capacity*, the desired initial capacity for the underlying array, creates an array with the given capacity that uses the default equivalence tester. It throws an *IllegalArgumentException* when *capacity* < 0.

```
public Array(int capacity) {
    this(capacity, Objects.DEFAULT_EQUIVALENCE_TESTER);
}
```

11.3.2 Trivial Accessors

The *isEmpty* and *getSize* methods are inherited from the *AbstractCollection* class. Recall that *getCapacity* returns the current capacity of the collection.

```
public int getCapacity(){
    return a.length;
}
```

| *Correctness Highlights:* By definition, the capacity is the length of the underlying array. ■

The *read* method takes *p*, a valid user position, and returns the element at user position *p*. It requires that *p* is a valid position.

```
protected final E read(int p){
    return (E) a[getIndex(p)];
}
```

| *Correctness Highlights:* From the correctness of *getIndex*, u_p is correctly returned. ■

The public *get* method takes *p*, the desired user position, and returns the element at the position *p*. It throws a *PositionOutOfBoundsException* when *p* is not a valid position.

```
public E get(int p) {
    if (p < 0 || p ≥ size)
        throw new PositionOutOfBoundsException(p);
    return read(p);
}
```

| *Correctness Highlights:* By SIZE, the *PositionOutOfBoundsException* is thrown when the user position is out of range. The rest of the correctness follows from that of *read*. ■

11.3.3 Representation Accessors

While $a[i] = u_i$ for the Array class, data structures that extend Array do not always have $u_i = a[i]$. For example, a circular array has an instance variable, *start* such that $a[start] = u_0$. Thus, the abstraction function is different for the CircularArray. We introduce representation accessors *getPosition* and *getIndex* to translate between the user position and underlying index. We also introduce representation accessors, *nextIndex* and *prevIndex* to use to move forward or backwards within the underlying array. When the abstraction function changes, we must override these representation accessors. Calling these methods adds a small amount of overhead that could be removed for the Array class if the methods were inlined*.

The first representation accessor, *getIndex*, takes *p*, a valid user position, and returns the corresponding index in the underlying array *a*. To avoid unnecessary bound checks, this method requires *p* to be a valid position.

```
int getIndex(int p){
    return p;
}
```

Correctness Highlights: Follows from PLACEMENT. ■

The *getPosition* method takes *index*, an underlying array index that is in use, and returns the corresponding user position. This method requires *index* is in use.

```
int getPosition(int index){
    return index;
}
```

Correctness Highlights: Follows from PLACEMENT. ■

The method *nextIndex* takes *index*, an underlying index of *a*, and returns the underlying index for the next element in the collection. Note that this method requires that *index* is between 0 and $n - 2$, inclusive. It must be overridden by any data structure in which the underlying index and user position may differ.

```
int nextIndex(int index){
    return index+1;
}
```

Correctness Highlights: The correctness follows directly from PLACEMENT. Since *index* is required to be at most $n - 2$, the return value is always an index that is in use. ■

Similarly, we introduce *prevIndex* that takes *index*, an underlying index, and returns the underlying index for the previous element in the collection. Note that this method requires that *index* is between 1 and $n - 1$, inclusive. It must be overridden by any data structure in which the underlying index and user position may differ.

*The Java compiler will not inline these methods because they are not final.

```

int prevIndex(int index){
    return index-1;
}

```

Correctness Highlights: The correctness of this method also directly follows from PLACEMENT. Since *index* is required to be at least 1, the return value is always an index that is in use. ■

11.3.4 Algorithmic Accessors

The *findPosition* method is used by all of the methods that must locate an element in the collection. This method takes *value*, the element to be located. It returns the first position *p* in the collection such that *value* is equivalent to the element at position *p*, or *NOT_FOUND* if there is no equivalent element in the collection. To find the last occurrence of an element, a similar method could be written that starts at the back of the array.

```

protected int findPosition(E value){
    for (int pos=0; pos<size; pos++){
        if (equivalent(value, read(pos)))
            return pos;
    }
    return NOT_FOUND;
}

```

Correctness Highlights: Follows from PLACEMENT, SIZE, and the correctness of *read*. After all positions are considered, if the element has not been found then it is not in the collection. ■

The method *contains* takes *value*, the element to be located, and returns *true* if and only if an equivalent element exists in the collection.

```

public boolean contains(E value) {
    return (findPosition(value) != NOT_FOUND);
}

```

The public method *positionOf* takes *value*, the element to be located, and returns the position in the collection for the first occurrence (if any) of an element equivalent to *value*. It throws a *NoSuchElementException* when no element in the collection is equivalent to *value*.

```

public int positionOf(E value) {
    int loc = findPosition(value);
    if (loc == NOT_FOUND)
        throw new NoSuchElementException();
    return loc;
}

```

11.3.5 Representation Mutators

The internal *resizeArray* method takes *desiredCapacity*, the desired capacity of the underlying array, and changes the capacity of the underlying array to *desiredCapacity* while maintaining the same positional collection. It throws an *IllegalArgumentException* when executing it would make the array capacity too small to hold the current collection. To improve efficiency, this method uses the Java native method *System.arraycopy* where

System.arraycopy(srcArray, srcIndex, destArray, destIndex, num)

copies the first *num* elements from array *srcArray* starting at index *srcIndex* into the array *destArray* starting at index *destIndex*.

```
void resizeArray(int desiredCapacity){
    if (desiredCapacity < size)
        throw new IllegalArgumentException("desiredCapacity < size");
    a = moveElementsTo(new Object[desiredCapacity]);
}
```

Correctness Highlights: Since a new array is created when $\text{desiredCapacity} \geq \text{size}$, *CAPACITY* is preserved. Otherwise, an exception is appropriately thrown. The other properties are preserved by the following *moveElementsTo* method. ■

The method *moveElementsTo* to copies the elements from the old array, if any, to the new one. (The old array will not exist when *resizeArray* is called for initialization purposes by the constructor.) The method takes *newArray*, the array to which the elements should be moved, and returns the parameter value for convenience.

```
protected Object[] moveElementsTo(Object[] newArray) {
    if (a != null)
        System.arraycopy(a, 0, newArray, 0, size);
    return newArray;
}
```

Correctness Highlights: By *PLACEMENT*, the array indices in use are $a[0], \dots, \text{size}-1$. Those are the elements copied preserving *PLACEMENT*. *NONRETENTION* is maintained since Java initializes all slots in the newly allocated array to *null*. Observe that the position (internal index) for all existing elements does not change, so there is no need to invalidate the active markers. ■

The public method *ensureCapacity* takes *capacity*, the desired capacity, and increases the capacity of the underlying array if needed.

```
public void ensureCapacity(int capacity) {
    resizeArray(capacity);
}
```

The method *trimToSize* reduces the capacity of the array to the current number of elements in the collection.

```
public void trimToSize() {
    if (size != a.length)
        resizeArray(size);
}
```

Correctness Highlights: If $a.length = size$, then no change is needed. Otherwise, the correctness follows from that of *resizeArray* and *SIZE*. ■

11.3.6 Content Mutators

The simplest content mutator, *set*, takes p , a user position to update, and *value*, the element to put at position p . It returns the prior element at position p . After this method executes, $u_p = value$. It throws a *PositionOutOfBoundsException* when p is not a valid position.

```
public E set(int p, E value) {
    if (p < 0 || p ≥ size)
        throw new PositionOutOfBoundsException(p);
    int index = getIndex(p);
    E oldValue = (E) a[index];
    a[index] = value;
    return oldValue;
}
```

Correctness Highlights: By *SIZE* the exception is appropriately thrown. By the correctness of *getIndex*, *oldValue* references the element that was held at position p , and the collection is updated so $u_p = value$. Thus *PLACEMENT* is maintained, and the correct element is returned. ■

The method *swap* takes $pos1$, a valid position, and $pos2$, a valid position. It swaps the values held in positions $pos1$ and $pos2$. It throws an *PositionOutOfBoundsException* when either $pos1$ or $pos2$ is not a valid position.

```
public void swap(int pos1, int pos2){
    if (pos1 < 0 || pos1 ≥ size)
        throw new PositionOutOfBoundsException(pos1);
    if (pos2 < 0 || pos2 ≥ size)
        throw new PositionOutOfBoundsException(pos2);
    swapImpl(pos1, pos2);
    version.increment();
} //invalidate all markers for iteration
```

Correctness Highlights: After checking that $pos1$ and $pos2$ are valid positions, the corresponding array slots are swapped by *swapImpl*. The rest of the correctness follows from that of *swapImpl* which also invalidates all markers for iteration (unless $pos1 = pos2$). ■

To avoid unnecessary bounds checks, in internal methods, such as the sorting algorithms, we introduce an internal method *swapImpl* that takes $pos1$, a valid position, and $pos2$, a valid position. This method swaps the elements held in positions $pos1$ and $pos2$. It requires that $pos1$ and $pos2$ are both valid positions. Since this may cause the element being moved to be seen twice or not at all during iteration, any public method that calls it must invalidate all active markers.

```

void swapImpl(int pos1, int pos2) {
    if (pos1 == pos2) return;
    int index1 = getIndex(pos1);
    int index2 = getIndex(pos2);
    Object temp = a[index1];
    a[index1] = a[index2];
    a[index2] = temp;
}

```

Correctness Highlights: The correctness follows from PLACEMENT. ■

The *move* method takes *fromPos*, the current position of the element to move, and *toPos*, the position where the element is to be moved. It requires that *fromPos* and *toPos* are both legal positions.

```

void move(int fromPos, int toPos) {
    a[getIndex(toPos)] = a[getIndex(fromPos)];
}

```

The *put* method takes *p*, a valid position, and *value*, the object to place at position *p*. It requires that *p* is a legal position.

```

void put(int p, Object value) {
    a[getIndex(p)] = value;
}

```

Correctness Highlights: Follows from that of *getIndex*. Since both *value* and the prior element in position *p* are not persistent, iteration through any existing markers can properly continue. ■

Methods to Perform Insertion

Next we present an internal method that moves elements, as needed, to make a gap in the array for a new element to be inserted. Specifically, the *createGap* method takes *p*, a valid user position where the gap is to be created, and moves elements at positions *p*, ..., *size* - 1 to positions *p*+1, ..., *size*. For efficiency, we directly use the position as an index into the underlying array. Therefore, this method must be overridden by any data structure in which the underlying index and user position may differ.

```

protected void createGap(int p){
    System.arraycopy(a, p, a, p+1, size-p);
    version.increment();
}
//invalidate all active markers for iteration

```

Correctness Highlights: Follows immediately from that of *arraycopy*. Since elements are shifted left, some persistent element could be seen twice. Thus all active markers must be invalidated. ■

The method *addImpl* takes *p*, a valid user position, and *value*, the object to insert. It inserts object *value* at position *p* and increments the position number for the elements that were at positions

$p, \dots, size - 1$. It returns *null* for an untracked implementations. For tracked implementations, it will be overridden to return a tracker.

```

PositionalCollectionLocator<E> addImpl(int p, Object value) {
    if (size == a.length)
        throw new AtCapacityException(a.length);
    if (p != size)
        createGap(p); //invalidates active locators for iteration
    a[getIndex(p)] = value;
    size++;
    return null;
}

```

Correctness Highlights: The instance variable *size* is incremented to preserve SIZE. CAPACITY is maintained by throwing an exception if the array is already at capacity. We now argue that PLACEMENT is maintained. After *add* executes, the collection contains $size' = size + 1$ elements: $u_0, \dots, u_{size-1}$ and *value*. Elements $u_0, \dots, u_{p-1}$ are unchanged. By the correctness of *createGap*, elements $u_p, \dots, u_{size-1}$ are shifted to $a[p+1], \dots, a[size]$. By the correctness of *getIndex*, $a[p] = value$. Thus the resulting abstraction is

$$\langle u'_0, \dots, u'_{size'-1} \rangle = \langle u_0, u_1, \dots, u_{p-1}, value, u_{p+1}, \dots, u_{size-1} \rangle$$

as desired. Finally, NONRETENTION holds since all array positions that were not in use are not changed except for position *size* which is now in use. The active markers are invalidated in *createGap*, which is called unless $p = size$. ■

The public *add* method that takes *p*, a valid user position, and *value*, the new element, inserts *value* at position *p* and increments the position number for the elements that were at positions $p, \dots, size - 1$. More formally, after the operation, the user sequence $\langle u'_0, \dots, u'_{size'-1} \rangle$ is

$$\langle u_0, u_1, \dots, u_{p-1}, value, u_p, \dots, u_{size-1} \rangle.$$

It throws a *PositionOutOfBoundsException* when *p* is neither *size* nor a valid position, and it throws an *AtCapacityException* when the underlying array is already at capacity. Recall that this method is a critical mutator unless $p = size$.

```

public final void add(int p, E value) {
    if (p < 0 || p > size)
        throw new PositionOutOfBoundsException(p);
    addImpl(p, value);
}

```

Finally, the public *add* method from the *Collection* interface takes *value*, the new element, and inserts it at the end of the collection. It throws an *AtCapacityException* when the underlying array is already at capacity.

```

public void add(E value) {
    add(size, value);
}

```

Correctness Highlights: Follows from that of *add* and the fact that position *size* is the position just after the last position currently used. ■

Methods to Perform Deletion

Next we present an internal method that moves elements (if needed) to close a gap created when elements are removed. For efficiency, it directly uses the position as an index into the underlying array. Therefore method must be overridden by any data structure in which the underlying index and user position may differ.

The *closeGap* method takes *fromPos*, the first position in the gap to close, and *toPos*, the last position in the gap to close. This method moves the elements that were at positions *toPos* + 1, . . . , *size* - 1 to positions *fromPos*, . . . , *size* - 1 - (*toPos* - *fromPos* + 1). We use *Arrays.fill* to set all slots in the underlying array that are no longer in use to *null*. The call *Arrays.fill(array, fromIndex, toIndex, val)* sets indices *fromIndex*, . . . , *toIndex* - 1 to *val*.

```
protected void closeGap(int fromPos, int toPos){
    int numElementsToMove = size - toPos - 1;
    if (numElementsToMove != 0)
        System.arraycopy(a, toPos+1, a, fromPos, numElementsToMove);
    Arrays.fill(a, fromPos+numElementsToMove, size, null);
}
```

Correctness Highlights: The number of elements to move is

$$(size - 1) - (toPos + 1) + 1 = size - toPos - 1.$$

The call to *arrayCopy* moves the elements that were at positions *toPos* + 1, . . . , *size* - 1 to positions *fromPos*, . . . , *size* - 1 - (*toPos* - *fromPos* + 1). The last *toPos* - *fromPos* + 1 positions switch are no longer in use. Since

$$\begin{aligned} size - (toPos - fromPos + 1) &= size - toPos + fromPos - 1 \\ &= fromPos + numElementsToMove, \end{aligned}$$

setting indices *fromPos* + *numElementsToMove*, . . . , *size* - 1 to *null* satisfies the second part of the specification. ■

We now present the public methods that remove elements. The most general such method, *removeRange*, takes as input *fromPos*, a valid position, and *toPos*, a valid position. This method requires $0 \leq fromPos \leq toPos < size$. It removes the elements at positions *fromPos*, . . . , *toPos*, inclusive, from the collection and decrements the positions of the elements at positions *toPos* + 1 to *size* - 1 by *toPos* - *fromPos* + 1 (the number of elements being removed). More formally, after this method executes the user sequence $\langle u'_0, \dots, u'_{size'-1} \rangle$ is $\langle u_0, \dots, u_{fromPos-1}, u_{toPos+1}, \dots, u_{size-1} \rangle$. It throws a *PositionOutOfBoundsException* when either of the arguments is not a valid position, and it throws an *IllegalArgumentException* when *fromPos* is greater than *toPos*. This method is a critical mutator unless *toPos* = *size* - 1.

```
public void removeRange(int fromPos, int toPos) {
    if (fromPos < 0 || toPos ≥ size)
        throw new PositionOutOfBoundsException();
    if (fromPos > toPos)
        throw new IllegalArgumentException();
    closeGap(fromPos, toPos);
    size = size - (toPos - fromPos + 1);
    if (toPos != size - 1)
        version.increment();
}
```

//removed elements end at last element
//invalidate locators for iteration

Correctness Highlights: By PLACEMENT, $a[fromPos], \dots, a[toPos]$ exist and can be removed by *closeGap*. Since *size* is decremented by the number of elements removed, SIZE is maintained. Also, since *a.length* does not change and *size* is decreased, CAPACITY is preserved. By the correctness of *closeGap*,

$$\begin{aligned}\langle u'_0, \dots, u'_{size'-1} \rangle &= \langle u'_0, \dots, u'_{fromPos-1}, u'_{fromPos}, \dots, u'_{size-numRemoving-1} \rangle \\ &= \langle u_0, \dots, u_{fromPos-1}, u_{toPos+1}, \dots, u_{size-1} \rangle,\end{aligned}$$

so PLACEMENT preserved. NONRETENTION is also preserved by *closeGap*. Finally, the locators are invalidated exactly when the positions of all remaining elements are not changed. ■

The methods *remove*, *removeFirst*, *removeLast*, and *clear* all call the *removeRange* method. The *remove* method takes *p*, a valid position. It removes the element at position *p* and shifts elements $u_{p+1}, \dots, u_{size-1}$ left by one position. More formally, after this method executes the user sequence $\langle u'_0, \dots, u'_{size'-1} \rangle$ is $\langle u_0, \dots, u_{p-1}, u_{p+1}, \dots, u_{size-1} \rangle$. It returns the removed element u_p . It throws a *PositionOutOfBoundsException* when *p* is not a valid position. This method is a critical mutator unless the last element is the one being removed (i.e., $p = size - 1$).

```
public E remove(int p) {
    if (p < 0 || p ≥ size)
        throw new PositionOutOfBoundsException(p);
    E removedElement = get(p);
    removeRange(p, p);
    return removedElement;
}
```

Correctness Highlights: By the correctness of *get*, *removedElement* is the element previously at position *p*. The rest of the correctness follows from that of *removeRange*. Observe that unless $p = size - 1$, *removeRange* will invalidate the active markers. ■

The method *removeFirst* removes the element at position 0 and decrements the position for the elements that were at positions 1 to *size-1*. It returns the element that was removed, and throws a *NoSuchElementException* when the collection is empty. This method is a critical mutator unless the collection holds only a single element.

```
public final E removeFirst() {
    if (isEmpty())
        throw new NoSuchElementException("collection is empty");
    return remove(0);
}
```

Correctness Highlights: By definition, position 0 is the first position in the collection. The rest of the correctness follows from that of *isEmpty* and *remove*. Observe that unless $size = 1$, *removeRange* will invalidate the active markers. ■

The method *removeLast* removes the element at position $size - 1$, and returns the element that was removed. It throws a *NoSuchElementException* when the collection is empty.

```

public final E removeLast() {
    if (isEmpty())
        throw new NoSuchElementException(“collection is empty”);
    return remove(size-1);
}

```

Correctness Highlights: By *SIZE*, the last element of the collection is that at position *size-1*. The rest of the correctness follows from that of *isEmpty* and *remove*. Since only the last element is removed, the markers will not be invalidated. ■

Finally, the *remove* method from the *Collection* interface takes *value*, the element to be removed, and removes the first element in the collection equivalent to *value*. It returns *true* if and only if an element is removed. This method is a critical mutator unless the last element is removed.

```

public boolean remove(E value) {
    int position = findPosition(value);
    if (position == NOT_FOUND)
        return false;
    removeRange(position, position);
    return true;
}

```

Correctness Highlights: The correctness follows from that of *findPosition* and *removeRange*. ■

The *clear* method removes all elements from the collection. Unless the collection is empty, this method is a critical mutator.

```

public void clear(){
    if (!isEmpty()) {
        Arrays.fill(a, 0, size, null);
        size = 0;
        version.increment();
    }
}

```

Correctness Highlights: Setting slots $0, \dots, size - 1$ to *null* preserves *NONRETENTION*. *SIZE* is preserved since after this method completes, *size=n=0*. *CAPACITY* holds since *resizeArray* ensures that *a.length* ≥ 0 . Finally, *PLACEMENT* holds vacuously since the collection is empty. ■

The *traverseForVisitor* method takes *v*, a visitor to apply for each element in the collection. This method traverses the entire collection on behalf of *v*. It throws an *Exception* when the visitor throws an exception. We override the implementation from *AbstractCollection* for efficiency. Recall that on advantage of using a visitor, instead of an iterator, is to avoid garbage collection overhead since no extra object needs to be created to support iteration.

```

public void traverseForVisitor(Visitor<? super E> v) throws Exception {
    for (int p = 0; p < size; p++)
        v.visit((E) get(p));
}

```

Correctness Highlights: Follows from the PLACEMENT and SIZE properties. ■

11.3.7 Locator Initializers

The *iterator* method creates a new marker that starts at FORE.

```
public PositionalCollectionLocator<E> iterator() {
    return new Marker(FORE);
}
```

Correctness Highlights: Follows from the Marker constructor, and the fact that position -1 logically precedes the first position. ■

The *iteratorAtEnd* method creates a new marker that starts at AFT.

```
public PositionalCollectionLocator<E> iteratorAtEnd() {
    return new Marker(size);
}
```

Correctness Highlights: Follows from that of the Marker constructor, and the fact that position *size* logically follows the last position. ■

The *iteratorAt* method takes *pos*, the user position of an element. returns a new marker that is at the given position. It throws a *NoSuchElementException* when the given position is not a valid user position.

```
public PositionalCollectionLocator<E> iteratorAt(int pos) {
    if (pos < 0 || pos ≥ size)
        throw new NoSuchElementException();
    return new Marker(getIndex(pos));
}
```

Correctness Highlights: Follows from the correctness of the Marker constructor and the *getIndex* method. ■

Finally, the *getLocator* method takes *value*, the target, and returns a marker initialized to the position of the first element in the collection equivalent to *value*. It throws a *NoSuchElementException* when *value* does not occur in the collection.

```
public PositionalCollectionLocator<E> getLocator(E value) {
    int position = findPosition(value);
    if (position == NOT_FOUND)
        throw new NoSuchElementException();
    return new Marker(position);
}
```

11.4 Sorting Algorithms

Applications often involve sorting a positional collection. We present a variety of sorting algorithms. They all use the internal methods that convert between the position and underlying index so that they work without modification for array, circular array, dynamic array, or dynamic circular array.

All of the sorting algorithms can be called through a public method that takes no argument and uses the default comparator, or through a public method that takes a comparator as an argument. We provide this flexibility since some objects have multiple ways of defining the comparator, and it might be desirable to independently sort the elements in multiple ways, such as to sort a list of mail messages by date or by sender. Algorithms need this kind of support as well. For example, Shamos' divide-and-conquer algorithm for finding a closest pair of points in the plane [125], needs to both sort the points according to x -coordinate and according to the y -coordinate.

The TrackedArray Class (Chapter 14) also illustrates an example usage of this capability, by sorting an array of objects where each object contains both a reference to an element and a tracker for that element. This allows the trackers to be preserved when sorting a tracked array.

Sorting a pointer-based implementation of a positional collection could be accomplished by putting all list items in an array, sorting them, and rebuilding the list. However, this would require additional space and often more time. Therefore the SinglyLinkedList class (Chapter 15) illustrates how these sorting algorithms can be implemented directly in a linked list. The reader interested in a deeper understanding of a particular algorithm may wish to study the implementation in this chapter and then refer to Chapter 15 to see how the same algorithm can be customized for a pointer-based structure.

The following four orthogonal properties provide a useful way to categorize sorting algorithms. Observe that all but the last definition can apply to any type of algorithm, not just sorting algorithms.

- A **comparison-based algorithm** is any algorithm in which information about the relative order of elements is only obtained through comparing elements. It has been proven that any comparison-based sorting algorithm has asymptotic time complexity of at least $\Omega(n \log n)$ [59]. All of the sorting algorithms we discuss except for counting sort, radix sort, and bucket sort are comparison-based algorithms.
- A **randomized algorithm** is any algorithm in which the behavior varies based on the outcome of a random number generator. Randomized quicksort is the only randomized sorting algorithm that is competitive. All of the other sorting algorithms we present are **deterministic** meaning that for a fixed input, the execution will be the same every time.
- An **in-place algorithm** is an algorithm in which only a constant amount of space is needed beyond that used to hold the input. Both insertion sort and quicksort are in-place algorithms. Also, our implementation of merge sort for a linked list is an in-place algorithm. In contrast, merge sort for an array-based implementation is not an in-place algorithm since it requires a linear-sized auxiliary array during the sorting process.
- A **stable sorting algorithm** is a sorting algorithm in which the relative order of equivalent elements is preserved. Our implementations of counting sort, insertion sort, merge sort, and radix sort are all stable. In contrast, an array-based implementation of quicksort cannot be stable without destroying its performance.

Before providing the implementation of a variety of sorting algorithms, we discuss the advantages and disadvantages of each.

Insertion Sort is one of a large number of quadratic time sorting algorithms. Other quadratic time sorting algorithms include **shell sort**, **bubble sort**, and **selection sort**. The primary advantage of insertion sort is that it runs in linear-time when the collection is **nearly sorted**, meaning that each element is only a constant number of positions from its sorted order. Because of this property, one can often improve the speed of other sorting algorithms by using insertion sort during the final phases when the array is nearly sorted. We illustrate this use of insertion sort in our implementation of bucket sort (Section 11.4.7) and for our quicksort implementation for a singly linked list (Section 15.5). We present all other sorting algorithms in their “pure” form, where the one algorithm is used to complete the entire sorting process. In general, one can improve the efficiency of any recursive sorting algorithm by using insertion sort when the subarray is sufficiently small[†].

The basic approach taken by insertion sort is to sort the positional collection from the left to the right, moving the next element into its proper position by shifting the larger one forward one position. Our implementation of insertion sort is in-place for both the array-based and list-based representations.

Merge sort has worst-case $O(n \log n)$ time complexity. It uses a technique called **divide-and-conquer**. In general, a divide-and-conquer algorithm divides the problem into subproblems. Then each of the subproblems are recursively solved (unless they are small enough that some direct algorithm can be used). Finally, the solutions to the subproblems are combined to create a solution to the original problem. Merge sort divides the positional collection into two equal halves, recursively sorts each half, and then merges the two sorted halves. Observe that the $\Omega(n \log n)$ lower bound for comparison-based sorting algorithms implies that no comparison-based sorting algorithm can perform asymptotically better than merge sort.

The advantage of merge sort over other algorithms with worst-case $O(n \log n)$ time complexity is that it does not rely upon using an auxiliary data structure, such as a priority queue or ordered collection. A disadvantage of the array-based implementation of merge sort is that it must merge the elements into an auxiliary array and then copy them back to the original array, so it is not an in-place sorting algorithm. If worst-case $O(n \log n)$ time complexity is not required, then quicksort should be considered.

Heap sort is a worst-case $O(n \log n)$ comparison-based sorting algorithm that uses a priority queue (Chapter 24) to organize the elements. More specifically, all n elements are inserted into a priority queue that uses a reverse comparator (Section 5.2.3), and then *extractMax* is used to extract the n elements. This algorithm is called heap sort since typically the binary heap data structure (Chapter 25) is used as the *PriorityQueue* implementation.

Tree sort is another worst-case $O(n \log n)$ comparison-based sorting algorithm. It uses an ordered collection (Chapter 29) to organize the elements. Our implementation uses a red-black tree (Chapter 34). In particular, first all elements are inserted into the red-black tree (which takes $O(n \log n)$ time), and then a linear-time inorder traversal extracts the elements in sorted order.

Quicksort is the comparison-based sorting algorithm that runs the most efficiently in practice, especially when tuned. If comparisons are expensive, then either radix sort or bucket sort, which do not use any comparisons, should be considered. An advantage of quicksort, and part of the reason why it runs so quickly in practice, is that it is an in-place sorting algorithm. Like merge sort, quicksort is a divide-and-conquer algorithm. In quicksort, the elements are

[†]The best choice for the cutoff is the size of the collection for which, on average, insertion sort is faster than the primary sorting algorithm.

partitioned into two portions where all elements in left portion are less than or equal to the elements in the right portion. Then the two portions are recursively sorted. A disadvantage of quicksort is that it has worst-case quadratic time (for example, when the array is already sorted). There are two ways in which such bad behavior can be avoided in practice.

- One way is to pick a random element to partition around at each recursive call. This method, **randomized quicksort**, guarantees an expected time complexity of $O(n \log n)$. Furthermore, the expectation is taken only over the randomization used within the algorithm. In other words, while the worst-case cost for randomized quicksort is still quadratic, such poor behavior is very unlikely regardless of the input.
- A second option is to partition around the median of the first, last, and middle element of the subcollection being sorted. This technique, called the **median-of-three** method, is the approach taken here. While the median-of-three method still has worst case quadratic time, in practice it works as well as randomized quicksort and does not need to use a random number generator. However, unlike randomized quicksort there is an input that will cause the worst-case quadratic behavior.

Another common variation used with quicksort to further speed up the running time is to only recursively apply quicksort on subarrays of size about 8 or larger. Upon completion, the array will be nearly sorted and a single final pass of insertion sort is used to complete the sorting process. (Insertion sort could instead be used to sort all subarrays of size less than 8, but it is more efficient to use a single pass of insertion sort at the end.)

There are two different partitioning methods (and many variations thereof) commonly used. The first is the partition method originally proposed by Hoare [82] that guarantees that at most $n/2$ swaps occur, and that performs well even when there are many equivalent elements in the collection. The other partition method, proposed by Lomuto, has the advantage that it is simpler to properly implement [23] and only requires the ability to iterate forward, which makes it a good choice for a singly linked list (see Chapter 15). However, the Lomuto partition could perform as many as n swaps. We implement Hoare's partition method in the `Array` class and Lomuto's partition method in the `SinglyLinkedList` class.

Our list-based quicksort implementation is stable. Therefore, it would support two successive sorts using different comparators, with the final collection being primarily sorted by the second comparator, and secondarily sorted (to break ties), by the first comparator.

Counting Sort is an alternative to comparison-based sorting algorithms that can improve performance when the elements being sorted are drawn from a small known set of possible values. More specifically, counting sort assumes that the elements to be sorted take on b distinct values which are mapped to the integers $0, \dots, k - 1$, where the smallest element maps to 0, the second smallest maps to 1, etc. The implementation we provide for counting sort is a stable sorting algorithm. The basic approach of counting sort is to take a first pass through the collection to count the number of occurrences of each element. From this, it is possible to compute the final location of all elements. In a final pass through the collection, all elements are put in the correct place. The time complexity for counting sort is $O(n + k)$. So, if $k = O(n)$ then this is a linear time sorting algorithm. To provide a stable sorting algorithm for use in radix sort (described next), our implementation is not in-place. If a non-stable sort is adequate, then an in-place implementation of counting sort is possible. We do not directly implement counting sort, but rather treat it as a special case of radix sort in which the base is k , so all the elements are treated as a one digit number.

Radix Sort is an extension of counting sort that breaks up each element into digits where each digit is mapped to an integer between 0 and $b - 1$, where b is the **base**. To allow our implementation of radix sort to apply to a wide range of objects, we have the application provide

a digitizer (see Section 5.3). Our design will work for any type of element for which a digitizer can be provided. Radix sort applies counting sort over the digits from the least to most significant digit. The time complexity for radix sort is $O(d(n + b))$ where d is the number of digits for each element.

Bucket Sort is another alternative to a comparison-based sorting algorithm that can improve performance if a good model for the distribution of the data is known. While often bucket sort is implemented to assume the elements are doubles in $[0, 1)$, we allow the application to provide a bucketizer (see Section 5.4) to discretize the elements into arbitrarily sized user-defined buckets. Ideally, the range of possible elements is divided into n buckets, each with an equal probability of holding a randomly selected element. Similar to counting sort, first the elements are placed into the appropriate bucket. Then, since each bucket is only expected to contain a constant number of elements, the array is nearly sorted, allowing insertion sort to complete the sorting in linear time. Bucket sort has a worst-case $O(n^2)$ performance that occurs when the elements are heavily clustered in the regions corresponding to just a few buckets.

11.4.1 Insertion Sort

Insertion sort works by first sorting the first 2 elements, then the first 3 elements, and so on until all n elements are sorted. Once the first j elements are sorted, element $j + 1$ can be put into the right place, by moving all elements that are larger than element $j + 1$ forward in the array. An execution of insertion sort is illustrated in Figure 11.2. The shaded portion of the array contains the elements from positions j to $n - 1$ (which are not yet processed). At each iteration of the for loop, the element in position j is moved into the correct position by shifting elements forward until an element less than or equal to the original element from position j is reached.

The *insertionsort* method with no arguments uses the default comparator to order the elements.

```
public void insertionsort() {
    insertionsort(Objects.DEFAULT_COMPARATOR);
}
```

The *insertionsort* method with an argument takes *comp*, the comparator to use to order the elements.

```
public void insertionsort(Comparator<? super E> comp) {
    insertionsortImpl(comp);
}
```

Correctness Highlights: The correctness follows from that of *insertionsortImpl*, which invalidates all active markers for iteration. ■

The internal *insertionsortImpl* method that takes *sorter*, the comparator to use, is our implementation of insertion sort.

```
void insertionsortImpl(Comparator sorter) {
    if (getSize() > 1) {
        for (int j = 1; j < getSize(); j++) {
            Object value = read(j);
            int i = j - 1;
            while (i ≥ 0 && sorter.compare(read(i), value) > 0) {
                move(i, i+1);
            }
        }
    }
}
```

//pos 0...j-1 are sorted
//holds object being put in place
//start at element at position just before it
//while element i larger
//move it forward

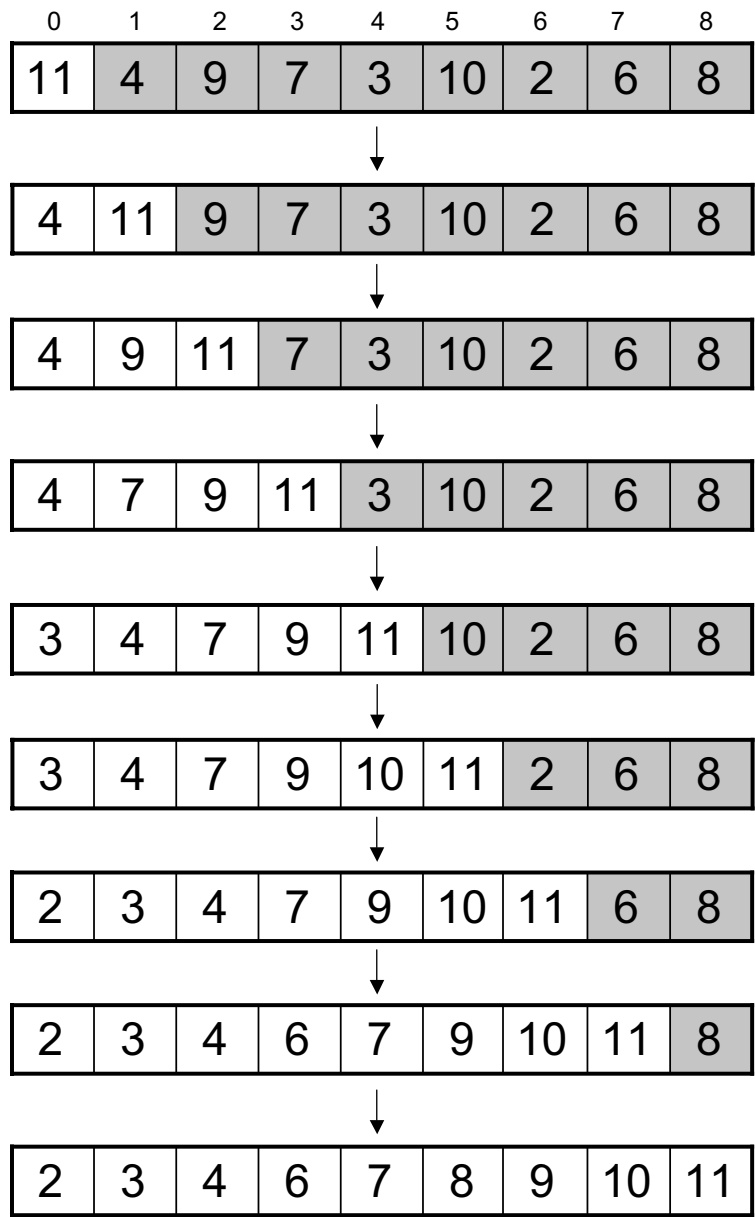


Figure 11.2
An illustration of insertion sort when run on an array of nine elements. The shaded portion of the positional collection is the portion that has not yet been processed. At each iteration of the algorithm, the leftmost shaded element is placed in its correct position after shifting the larger elements to the right by one place.

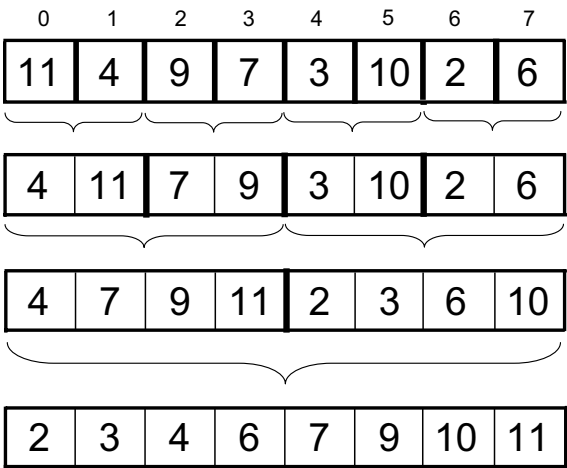


Figure 11.3 An illustration of merge sort when run on an array of eight elements. The top array shown is the original array, and the bottom row shown the sorted array. At each level of recursion the subarrays independently considered are marked by the solid black lines. The braces are used to show which subarrays are merged at the next level.

```
        i--;                                //move to the left
    }
    put(i+1, value);                          //put object into place
}
version.increment();                          //invalidate active markers for iteration
}
```

Correctness Highlights: First, clearly this is correct when $n \leq 1$ since no computation is needed. The rest of the correctness can be proven using a loop invariant that the subarray from positions 0 to j is sorted. This holds when $j = 0$. We now assume that the elements from positions 0 to $j - 1$ are sorted at the start of the for loop, and prove the elements in positions $0, \dots, j$ are sorted after the next execution of the for loop. It is easily checked that the element that was initially in position j is inserted in the correct order (and the other elements are kept in the same relative order). The check that $i \geq 0$ ensures that the correct behavior occurs when the element that began at position j is the smallest among those from indices 0 to j . The rest of the correctness follows from that of the *getSize*, *getPosition*, *get*, *move*, and *put* methods. ■

We briefly analyze the time complexity. In the worst case $O(j)$ time is spent at each iteration of the for loop. Thus, the asymptotic time complexity is at most $\sum_{j=1}^{n-1} (c \cdot j) = c \cdot n(n-1)/2 = O(n^2)$. It is easily verified that if the array is in reverse sorted order then this cost is incurred. Finally, observe that if each element is only a constant number of positions out of place, then each iteration of the for loop takes constant time, leading to an asymptotic cost of $O(n)$.

11.4.2 Mergesort

Next we present the merge sort algorithm which divides the array into two halves, recursively sorts each half and then merges the two sorted halves. This process is illustrated in Figure 11.3. The *mergesort* method with no arguments uses the default comparator to order the elements.

```

public void mergesort() {
    mergesort(Objects.DEFAULT_COMPARATOR);
}

```

The other *mergesort* method takes *comp*, the comparator that defines the ordering of the elements.

```

public void mergesort(Comparator<? super E> comp) {
    mergesortImpl(comp);
}

```

The *mergesortImpl* method that takes *sorter*, the comparator to use, starts the recursive merge sort method. Since merge sort requires an auxiliary array during the merge process, for efficiency we must allocate a second array of n elements to pass to the recursive method. Also, to simplify the recursive method, we temporarily move the elements so that position i is stored at index i (even when the underlying index is different from the user position). Putting these two design decisions together leads to the following design. We allocate a Java array *toSort* and move all elements into it so that *toSort*[i] references the element in position i . Then, during merge sort, *a* is used as the auxiliary array. Once merge sort has completed, the elements are moved back into *a* so that the original underlying index for position p is restored.

```

void mergesortImpl(Comparator<? super E> sorter) {
    if (getSize() > 1) {
        Object toSort[] = new Object[getSize()];           //array created to sort
        for (int i = 0; i < getSize(); i++)                 //put elements into toSort
            toSort[i] = read(i);                             //so pos i in index i
        mergesortImpl(toSort, a, 0, getSize()-1, sorter);   //sort toSort, using a as aux. array
        for (int i = 0; i < getSize(); i++)                 //put elements back into a
            put(i, toSort[i]);
        version.increment();                                //invalidate active markers for iteration
    }
}

```

Correctness Highlights: By the correctness of *getPosition* the elements are placed into *toSort* so that position i is at index i . Likewise, by the correctness of *getPosition* the sorted collection is placed back into *a* using the original mapping between positions and indices. The rest of the correctness follows from the recursive *mergesortImpl* method, after which all active markers are invalidated for iteration. ■

We now describe the recursive *mergesortImpl* method that takes *data*, the array to sort, *aux*, an auxiliary array to use for the merge, *left*, the starting index of the subarray to sort, *right*, the ending index of the subarray to sort, and *sorter*, the comparator to use to compare elements. It sorts the subarray from *left* to *right* using the given comparator. Dividing the subarray into two halves just involves finding the average of the left and right indices. Then the two halves are sorted recursively. We now discuss how the two sorted halves are merged. There is an index i for the current position of the left half of *data*, an index j for the current position of the right half of *data*, and the next position k to fill in *aux*. Until the end of either the left or right half of *data* is reached, whichever elements is smaller between *a*[i] and *a*[j] is copied into the next slot of *aux*, and the appropriate indices are incremented. Finally, the remainder of the left half of *data*, if any, is copied into *aux*, and finally all used portions of *aux* are copied back onto *data*.

```

void mergesortImpl(Object[] data, Object[] aux, int left, int right,
                    Comparator<? super E> sorter) {
    if (left < right) {
        int mid = (left + right) / 2;                //find the middle
        int i = left;                                //index in data of current loc in left half
        int j = mid+1;                                //index in data of current loc in right half
        int k = left;                                //index of current location in aux
        mergesortImpl(data, aux, left, mid, sorter); //recursively sort the left half
        mergesortImpl(data, aux, mid + 1, right, sorter); //recursively sort the right half
        while (i ≤ mid && j ≤ right) {                //merge two sorted halves
            if (sorter.compare((E) data[i], (E) data[j]) < 0) //if next element on left smaller
                aux[k++] = data[i++];                //move into aux
            else                                     //otherwise next element on right smaller
                aux[k++] = data[j++];                //move it into aux
        }
        System.arraycopy(data, i, aux, k, mid-i+1); //copy rest of left half to aux
        System.arraycopy(aux, left, data, left, j-left); //copy used portion of aux back into data
    }
}

```

Correctness Highlights: When $left \geq right$ the portion of *data* to sort has at most one element, so no computation is required to sort it. The rest of the correctness argument uses induction. By the inductive hypothesis it follows that after the two recursive calls to *mergesortImpl*, the left and right halves of *data* are sorted. The remainder of the proof relies on the fact that the two sorted subarrays are correctly merged, which is easily seen to be the case. ■

We now briefly discuss the time complexity analysis for merge sort. It can be seen that when the array has only one element it takes constant time, and that the merge itself takes linear time. To determine the time complexity of merge sort you can express a **recurrence equation** that $T(1) = \Theta(1)$ and $T(n) = 2T(n/2) + \Theta(n)$ where $T(n)$ is the time required for merge sort to sort n elements. This recurrence equation is obtained by observing that if the original subarray has n elements there are two recursive calls made on subarrays of size[‡] $n/2$ with $\Theta(n)$ time used for all other computation (besides the recursive calls). It can be shown that for this recurrence $T(n) = \Theta(n \log n)$. More specifically if $T(1) = 1$ and $T(n) = 2T(n/2) + cn$ (for n a power of 2), then $T(n) = cn \log_2 n + n$. See Appendix B for a description of a simple method to solve a recurrence equation such as the one here.

11.4.3 Heap Sort

In heap sort, all elements are added to a priority queue (Chapter 24). Then *extractMax* really removes the smallest element. Finally, *extractMax* is called n times to extract the elements in reverse sorted order. In the implementation we provide, a binary heap (Chapter 25) is used since it is very space efficient and the methods needed are those for which it is well suited. Several methods are designed to facilitate the extension for a tracked array, in which the nodes themselves are sorted instead of sorting the elements.

[‡]Technically one array is of size $\lfloor n/2 \rfloor$ and the other is of size $\lceil n/2 \rceil$ but in determining the asymptotic solution, these can be removed.

The *heapsort* method with no arguments sorts this collection with heap sort using the default comparator.

```
public void heapsort() {
    heapsort(Objects.DEFAULT_COMPARATOR);
}
```

The *heapsort* method that takes *comp*, a comparator, sorts this collection with heap sort using the provided comparator.

```
public void heapsort(Comparator<? super E> comp) {
    heapsortImpl(comp);
}
```

Correctness Highlights: The correctness follows from that of *heapsortImpl*, which invalidates all active markers for iteration. ■

The *buildPriorityQueue* method takes *pq*, the priority queue used by heap sort. It inserts all the elements in the positional collection on which this method is called into *pq*. This method is an example usage of the template design pattern. The *TrackedArray* class will override it to insert the nodes holding the elements into the priority queue.

```
void buildPriorityQueue(PriorityQueue<Object> pq) {
    pq.addAll(this);
}
```

The *heapsortImpl* method that takes *sorter*, the comparator to use, is the implementation of heap sort. While any *PriorityQueue* ADT implementation could be used, typically the binary heap is used, which is why this algorithm is known as heap sort.

```
void heapsortImpl(Comparator sorter) {
    PriorityQueue<Object> heap = new BinaryHeap<Object>(getSize(), sorter);
    buildPriorityQueue(heap);                                //put all elements in heap
    for (int i = getSize()-1; i ≥ 0; i--)                    //extractMax putting elements back into this
        put(i, heap.extractMax());                        //positional collection starting at position 0
    version.increment();                                    //invalidate active markers for iteration
}
```

Correctness Highlights: Follows directly from the correctness of the binary heap methods used, and the fact that the elements are extracted and placed in the array from greatest to smallest. ■

As discussed in Chapter 25, *addAll* takes linear time for a binary heap. Since *extractMax* is called *n* times to extract the elements in sorted order, and each call to *extractMax* takes logarithmic time, the overall time complexity is $O(n \log n)$.

If desired, it would be possible to implement the heapsort algorithm in place. First since a binary heap (Chapter 25) wraps an array, the array being sorted could be directly used. The binary heap *addAll* uses a linear time procedure to convert an arbitrary array in to a binary heap. This algorithm could be directly applied to the original array. The binary heap *extractMax*, when applied to a heap of size *size*, uses the subarray defined by slots 0, ..., *size* - 1. It places the highest priority (i.e., largest) element into slot *size* - 1, and then modifies slots 0, ..., *size* - 2 into a valid heap and decrements *size*. So after repeatedly applying *extractMax* to a heap, the resulting array will contain the elements in sorted order.

11.4.4 Tree Sort

Tree sort is similar to heap sort except that it uses a red-black tree (Chapter 34) data structure which implements the OrderedCollection ADT (Chapter 29). Tree sort first inserts all elements (which takes $O(n \log n)$ time) into a red-black tree and then performs a linear-time inorder traversal (using the visitor) to extract the elements in sorted order. The disadvantage of tree sort, relative to heap sort, is that the binary heap can support the required methods more efficiently than the red-black tree, which is designed to support a general-purpose search that is not needed here.

The *treesort* method with no arguments sorts this collection using the tree sort algorithm and the default comparator.

```
public void treesort() {
    treesort(Objects.DEFAULT_COMPARATOR);
}
```

The *treesort* method that takes *comp*, a comparator, sorts this collection using the tree sort algorithm and the provided comparator.

```
public void treesort(Comparator<? super E> comp) {
    treesortImpl(comp);
}
```

Correctness Highlights: The correctness follows from that of *treesortImpl*, which invalidates all active markers for iteration. ■

The *treesortImpl* method takes *sorter*, the comparator to use, is the implementation of tree sort. It creates an anonymous visitor that places each element into the array, in order, when called by the recursive *accept* method of the red-black tree. An iterator could be used instead of the visitor, but the iterator requires more computation to navigate the tree than the recursive *accept* method does.

```
void treesortImpl(Comparator<? super E> sorter) {
    RedBlackTree<E> tree = new RedBlackTree<E>(sorter);
    for (int i = 0; i < getSize(); i++)                //add all elements to tree
        tree.add(read(i));
    tree.accept(new Visitor<Object>() {                //use visitor to traverse
        int pos = 0;                                    //position for next element
        public void visit(Object o) throws Exception {
            put(pos++, o);
        }
    });
    version.increment();                               //invalidate active markers for iteration
}
```

Correctness Highlights: Follows directly from the correctness of the red-black tree methods used, and the fact that the elements are extracted and placed in the array from smallest to largest. ■

11.4.5 Quicksort

Like merge sort, quicksort is a divide-and-conquer algorithm. Recall that merge sort splits the array into two equal halves, recursively sorts the halves, and then merges the sorted subarrays. So merge

sort does very little computation before the recursive calls are made, and performs most of the required work after the recursive calls have completed. In contrast, quicksort performs all of its extra computation prior to making the recursive calls. Specifically, quicksort first selects a **pivot** element and subdivides the array with respect to the pivot element so that when the pivot element is in position p , all elements in positions $0 \leq i \leq p - 1$ are no larger than the pivot element, and all elements in positions $p + 1, \dots, n - 1$ are at least as large as the pivot element. Quicksort then completes the sort by recursively sorting the subarrays on both sides of the pivot element.

One variation in different implementations of quicksort is the mechanism used to select the pivot element. The simplest option is to just pivot around the last element in the subarray being considered. However, if the array is sorted (or reverse sorted) this leads to a worst case quadratic behavior. To avoid this bad behavior here are two approaches generally used:

- Select a random position uniformly between *left* and *right*, and, prior to calling *partition*, swap the randomly selected element with the one in position *right*. When using this partition method the resulting algorithm is called **randomized quicksort**.
- Find the position which is the median among the elements in the three positions, *left*, *right*, and the middle position, and swap this **median-of-three** element with the one in position *right*.

Two advantages of the median-of-three approach is that when the array is in sorted order it yields the best-case behavior, and it removes the cost associated with a random number generator. Without randomization, quicksort has worst-case quadratic time complexity. However, in practice using the median-of-three method yields the fastest implementation, so the median-of-three implementation is presented here.

The *quicksort* method with no arguments sorts this collection with quicksort using the default comparator.

```
public void quicksort() {
    quicksort(Objects.DEFAULT_COMPARATOR);
}
```

The *quicksort* method that takes *comp*, a comparator, sorts this collection with quicksort using the provided comparator.

```
public void quicksort(Comparator<? super E> comp) {
    if (getSize() > 1)
        quicksortImpl(0, getSize()-1, comp);
    version.increment(); //invalidate all markers for iteration
}
```

Correctness Highlights: The correctness follows from that of *quicksortImpl*. Since elements in the array are reordered, it is necessary to invalidate all active markers. ■

The recursive *quicksort* method takes *left*, the leftmost position within the subarray to be sorted, *right*, the rightmost position with the subarray, and *sorter*, the comparator to use when comparing the elements. It sorts the subarray from positions *left* to *right*, inclusive.

```
void quicksortImpl(int left, int right, Comparator<? super E> sorter){
    if (left < right) { //done when left >= right
        swap(getMedianOfThree(left, right, sorter), right);
        int mid = partition(left, right, sorter); //partition around right element
        quicksortImpl(left, mid-1, sorter); //recursively sort portion before pivot
    }
```

```

        quicksortImpl(mid+1, right, sorter);           //recursively sort portion after pivot
    }
}

```

Correctness Highlights: When $left \geq right$ the subarray has at most one element, so it is sorted. We now consider when $left < right$. By the correctness of *getMedianOfThree* and *swap*, the median of the left, middle, and right elements is placed as the right most element. By the correctness of *partition* all elements from positions *left* to *mid-1* are at most as large as the element in position *mid*, which is the pivot element. Likewise, the elements in positions *mid+1* to *right* are greater than the element in position *mid*. Thus, using an inductive argument (on the size of the subarray), it follows that once the two recursive calls complete, that the subarray from positions *left* to *right* will be sorted. ■

The method *getMedianOfThree* takes *left*, the leftmost position within the subarray, *right*, the rightmost position with the subarray, and *sorter*, the comparator to use when comparing the elements. It returns the position in the array of the median of the left, right, and middle element.

```

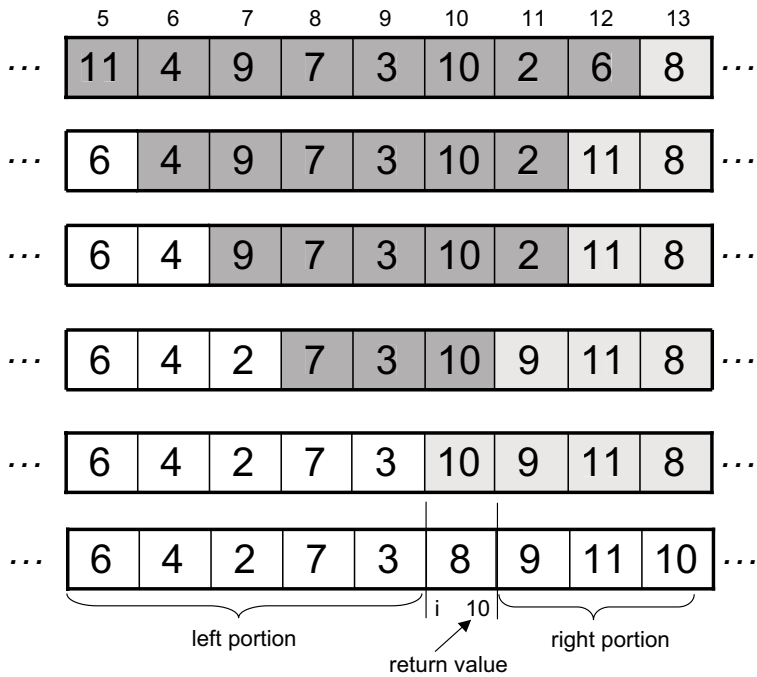
int getMedianOfThree(int left, int right, Comparator<? super E> sorter) {
    if (right - left + 1 ≥ 3) {                               //only perform if at least 3 elements in subarray
        int mid = (left + right)/2;                             //position of middle element of subarray
        E leftObject = read(left);                             //object at left
        E midObject = read(mid);                               //object at mid
        E rightObject = read(right);                           //object at right
        if (sorter.compare(leftObject, midObject) ≤ 0) {
            if (sorter.compare(midObject, rightObject) ≤ 0)
                return mid;
            else if (sorter.compare(rightObject, leftObject) ≤ 0)
                return left;
            else if (sorter.compare(midObject, rightObject) > 0) {
                return mid;
            }
        }
    }
    return right;
}

```

Correctness Highlights: It is easily verified that the position returned is the position for the median of the elements in positions *left*, *right*, and $mid = \lfloor (left + right)/2 \rfloor$. ■

The *partition* method takes *left*, the leftmost index of the subarray to partition, *right*, the rightmost index of the subarray to partition, and *sorter*, the comparator to use to order the elements. This method returns the final position of the pivot element, which is initially in position *right*. It modifies the array so that all elements in the subarray left of the returned position are strictly less than the pivot element, and all elements in the subarray right of the returned position are greater than or equal to the pivot element. See the bottom diagram in [Figure 11.7](#).

We use the partition method introduced by Hoare [82]. It pivots around the element *pivot* that begins in position *right* of the subarray. It uses two local variables to mark the ends of the processed portions of the array. Specifically *i*, which starts at position *left*, marks the end of the left portion holding elements less than *pivot*, and *j*, which starts at the position just before the position of the pivot, marks the start of the right portion holding elements not less than *pivot*. Until *i* and *j* cross

**Figure 11.4**

The modification made when calling *partition* on the array shown above when *left* is 5 and *right* is 13. The array is shown just before and just after each call to the *swap* method. The unshaded portion contains elements known to be $< \text{pivot}$, the darkly shaded portion are the unprocessed elements, and the lightly shaded portion (up to position *right* - 1) are the processed elements known to be $\geq \text{pivot}$. Note that after the left portion and right portion of the last subarray are recursively sorted, the subarray will be sorted.

(i.e., $i > j$), it repeatedly moves i forward to some element larger than the pivot, and moves j backwards to some element smaller than the pivot. Then the elements at positions i and j are swapped. Finally, the pivot element is swapped with the element at position i and its position is returned. Figure 11.4 shows a sample execution. Because of the swap performed by *partition*, quicksort is not a stable sorting algorithm.

```

int partition(int left, int right, Comparator<? super E> sorter){
    E pivot = read(right);                                //pivot around the right element
    int i = left;                                           //positions left...i-1 hold elements < pivot
    int j = right;                                          //positions j...right-1 hold elements >= pivot
    while (i < j){
        while (i < j && sorter.compare(read(i), pivot) < 0)
            i++;
        while (j > i && sorter.compare(read(j), pivot) ≥ 0)
            j--;
        if (i < j)
            swapImpl(i, j);
    }
    swapImpl(i, right);
    return i;
}

```

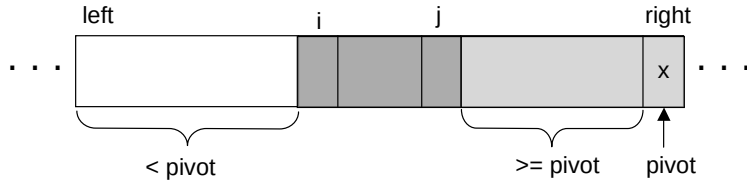



Figure 11.5

This figure illustrates the invariant maintained by the partition method. The elements in the unshaded regions must be $< pivot$, and the elements in the lightly shaded region must be $\geq pivot$. No claim is made about the elements in the darkly shaded region.

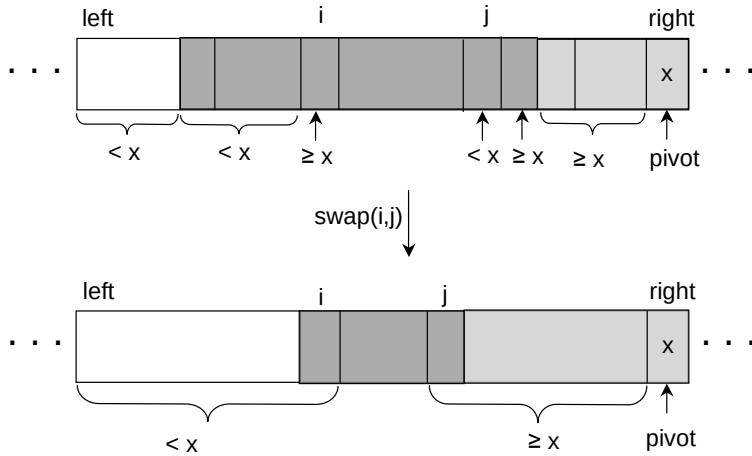


Figure 11.6

This figure illustrates the inductive step of the proof that the loop invariant is maintained.

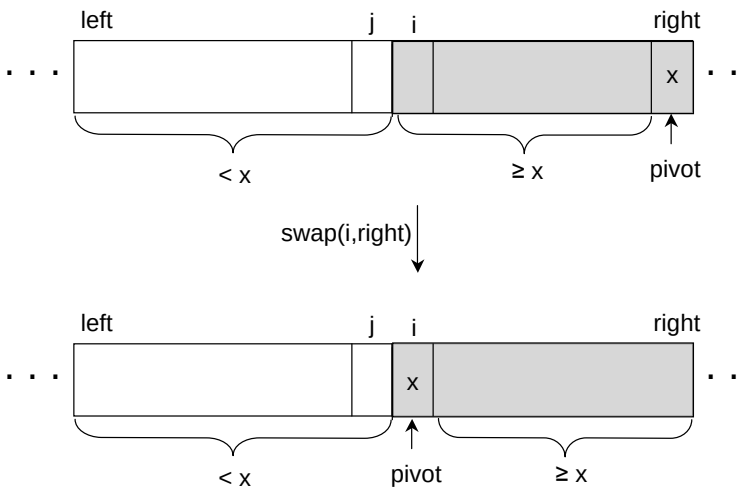


Figure 11.7

This figure illustrates the situation before and after $swap(i, right)$ is executed.

Correctness Highlights: We let e_p denote the element held in position p . That is, $e_p = \text{get}(p)$. We now prove that the following proposition is a loop invariant for the outer while loop:

$$(\forall \text{ left} \leq k < i, e_k < \text{pivot}) \wedge (\forall j < k < \text{right}, e_k \geq \text{pivot}).$$

Figure 11.5 visually shows this invariant.

The invariant vacuously holds before the loop is entered since all elements are between $e_i, \dots, e_j$ (for which the invariant makes no claim). We now prove that the loop invariant is maintained when the loop body is executed. Figure 11.6 illustrates this portion of the correctness argument. First observe that i is only incremented when $e_i < \text{pivot}$. Combined with the invariant it follows that:

$$\forall \text{ left} \leq k < i, e_k < \text{pivot}, \text{ and } e_i \geq \text{pivot}.$$

Similarly, since j is only decremented when $e_j \geq \text{pivot}$, combined with the invariant it follows that

$$\forall j < k < \text{right}, e_k \geq \text{pivot}, \text{ and } e_j < \text{pivot}.$$

There are two possible cases after the inner while loops are completed.

Case 1 ($i < j$): In this case, $\text{swap}(i, j)$ is executed. Combined with the above claims, after the positions of e_i and e_j are swapped, it follows that

$$(\forall \text{ left} \leq k \leq i, e_k < \text{pivot}) \wedge (\forall j \leq k < \text{right}, e_k \geq \text{pivot})$$

which is stronger than the stated loop invariant.

Case 2, ($i > j$): By the loop invariant, all elements in the subarray left of position i are $< \text{pivot}$, so j could not move further than the position immediately left of i . Likewise, since all elements in the subarray right of position j are $\geq \text{pivot}$, j could not move further than the position immediately right of i . Thus this case only occurs when $i = j + 1$, and from the above claims it immediately follows that the invariant holds.

Finally, we consider the result of executing $\text{swap}(i, \text{right})$ that occurs after the outer while loop has completed. This final step is illustrated in Figure 11.7. As in Case 2 above, it follows that when the loop exits, $i = j + 1$. From the invariant it follows each of $e_{\text{left}}, \dots, e_j$ are $< \text{pivot}$ and each of $e_i, \dots, e_{\text{right}-1}$ are $\geq \text{pivot}$. Also recall that $e_{\text{right}} = \text{pivot}$. Thus swapping e_i and e_{right} guarantees that the subarray is sorted since all elements in the subarray to the left of the final position for the pivot are less than it, and all elements in the subarray to the right of the final position for the pivot are at least as large as it. Finally, observe that the final position of the pivot (i) is returned. ■

We now briefly discuss the time complexity analysis for quicksort. The worst case for quicksort, even when randomly selecting each pivot, occurs when the pivot at each recursive call is either the smallest or largest element in the subarray. When this situation occurs, the size of the subproblems are $n, n-1, n-2, \dots, 1$ and linear time is spent for the partitioning of each recursive call. Thus the worst-case time complexity is

$$cn + c(n-1) + \dots + c = cn(n-1)/2 = O(n^2)$$

where c is some constant.

The expect-case time complexity of $O(n \log n)$ can be proven only for the randomized version of quicksort. However, a similar analysis holds for the median-of-three if the data are random. When using the median-of-three partition, there is still a specific input that causes quicksort to take quadratic time, but in practice its performance is like that for randomized quicksort [134].

We summarize the key aspects of the analysis of randomized quicksort as described by Cormen et al. [42]. The dominant cost of quicksort is the comparisons performed by the partition algorithm. In randomized quicksort, the pivot element is selected uniformly at random. The only aspect of the partitioning that affects the time complexity is the number of elements in each subarray that is recursively sorted. We use *left:right* to indicate that there are *left* elements in the subarray left of the partition element and *right* elements in the subarray right of the partition element. Since each of element in a subarray of size x is equally likely to be the pivot, it follows that each split of $\{0 : x - 1, 1 : x - 2, \dots, x - 2 : 1, x - 1 : 0\}$ occurs with probability $1/x$.

For ease of exposition let U_{ij} refer to the subarray of elements $u_i, \dots, u_j$ of the sort. Next introduce an indicator random variable X_{ij} which is 1 if u_i is compared to u_j , and otherwise is 0. Since every comparison involves the pivot element and the pivot element is not part of either recursive call, any two elements can be compared at most once. Also, observe that u_i and u_j are compared only if one of them is the first pivot chosen in U_{ij} which occurs with probability $\frac{2}{j-i+1}$ where $j - i + 1 = |U_{ij}|$. By linearity of expectation, the expected number of comparisons is

$$\sum_{i=1}^{n-1} \sum_{j=1}^{n-1} \frac{2}{j-i+1} \leq 2(n-1)(\ln n + 1) = O(n \log n).$$

Thus, the expected time complexity of randomized quicksort is $O(n \log n)$. An important aspect of this analysis is that no assumptions are made about the distribution of the elements to sort. The expectation is just over the random choices made in selecting the partition elements.

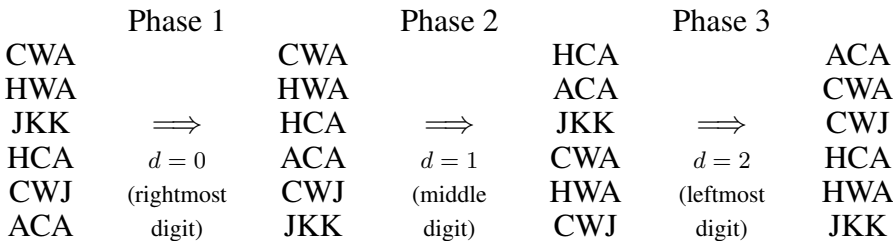
11.4.6 Radix Sort

Unlike the sorting algorithms described so far, radix sort does not directly take a comparator. Instead, the total order over the elements is implicitly defined by a provided *digitizer*. (See Section 5.3 for a discussion of the *Digitizer* interface and a sample implementation.) Specifically, a digitizer with base b will return an integer in $\{0, 1, \dots, b - 1\}$ for each “digit” of an element. One can thus treat each element as a base b number and the comparisons are defined relative to this.

The *radixsort* method takes *digitizer*, the digitizer to use, and sorts the collection with radix sort.

```
public void radixsort(Digitizer<? super E> digitizer) {  
    radixsortImpl(digitizer);  
}
```

As a simple example to illustrate radix sort, consider the task of sorting a set of initials. The digitizer would map A to 0, B to 1, and so on. Each element would be treated as a three digit, base 26 number. Suppose the input was {CWA, HWA, JKK, HCA, CWJ, ACA}. The execution of radix sort is illustrated in the diagram below.



In pass d , **counting sort** (a stable sorting algorithm) is used to sort the elements using digit d , where the least significant digit is digit 0. We now describe the execution of pass d . First, the

number of occurrences for each possible value of digit d is computed. Next these counts are added cumulatively so that $count[i]$ holds the number of elements whose digit d is no greater than i . These cumulative counts are then used to put each element in its proper position, with respect to the digit d , in a second array in such a way that elements with equal values in the digit d are kept in the same relative order.

The *radixsortImpl* method that takes *digitizer*, the digitizer to use, is the complete implementation of radix sort.

```
protected void radixsortImpl(Digitizer<? super E> digitizer) {
    Object[] from = new Object[getSize()];           //elements start in from
    Object[] to = new Object[getSize()];             //placed in sorted order into to
    int b = digitizer.getBase();                       //base for digitizer
    int count[] = new int[b];                         //counter for each possible value
    int numDigits = 0;                                //maximum number of digits in any element
    for (int i = 0; i < getSize(); i++) {
        from[i] = a[getPosition(i)];                  //move position i into index i of from
        numDigits = max(numDigits, digitizer.numDigits((E) from[i]));
    }
    for (int d = 0; d < numDigits; d++) {              //digit to use in current pass
        Arrays.fill(count, 0);                        //reset all counts to 0
        for (Object x : from)                          //count # elements with
            count[digitizer.getDigit((E) x, d)]++;    //each value for digit d
        for (int i = 1; i < b; i++)                    //update to cumulative count
            count[i] += count[i-1];
        for (int i = getSize()-1; i ≥ 0; i--)          //put elements in array to
            to[--count[digitizer.getDigit((E) from[i], d)]] = from[i];
        Object[] temp = from; from = to; to = temp;   //swap the "from" and "to" arrays
    }
    for (int i = 0; i < getSize(); i++)                //put sorted elements back into the collection
        put(getPosition(i), from[i]);
    version.increment();                               //invalidate locators for iteration
}
```

Correctness Highlights: We wish to prove the claim that after completion of pass d of the algorithm, the elements in the *to* array are sorted with respect to digits 0 through d . We argue by induction. The base case is trivial since there are no digits. Consider the contents of *from* and *to* after a given iteration d of the main loop, just before the *from* and *to* arrays are swapped. By the inductive hypothesis, we know that *from* contains the results of the previous pass (or the initial input if $d = 0$), satisfying the claim up through pass $d - 1$. To prove the inductive step, we need to argue that after pass d ,

1. all elements in *to* are sorted with respect to d , and
2. all elements with equal values of d are in the same relative order in *to* as they are in *from*.

We need to show that those with the smallest value of d are grouped together in the first part of the array, followed by those with the next larger value of d , and so on. It is easily verified that after the first inner *for* loop, the value of $count[v]$ is the number of elements with value v in digit d . Observe that the second inner *for* loop results in the new value for $count[v]$ being the sum $count[0] + \dots + count[v]$.

This determines the sizes of the groups. The ending index, then of each of the groups is exactly

the same of the size of the group, and the ending index of the previous group which is the series of values computes in *count* by the second inner loop. We now consider each element *from[i]* processed by the third inner loop, which starts from $i = n - 1$ and works towards $i = 0$. We argue that the third inner loop preserves the invariant that *count[v]* is the number of elements in *from[0], ..., from[i]* that have values in $\{0, 1, \dots, v\}$. Clearly this holds before the third inner loop begins executing. We now argue that each execution of the loop preserves the invariant. Let $x = \text{digitizer.getDigit}(\text{from}[i], d)$. By the invariant, there are *count[x]* elements with values $\{0, 1, \dots, x\}$, which in the sorted order would belong in *to[0], ..., to[count[x] - 1]*. Since there is now one less element in *from* with value *x*, *count[x]* is properly decremented by one. Therefore the invariant is preserved.

We now use the invariant of the third inner loop to show that the execution of that loop meets both conditions of the claim above. First, it is easily seen that each element with a given value for digit *d* is placed together in its proper group, according to the compute group sizes. Second, we observe that for each *i*, since *from[i]* appears last, it is properly placed at *to[count[x]-1]*, and since the count for its group is decremented by one, we know that all elements encountered with the same value of digit *d* are placed in their group in the same relative order, back to front.

Finally, we prove the correctness for radix sort by induction on *d*. Specifically, we prove that after processing digits $0, \dots, d$ that the elements are sorted with respect to those $d + 1$ digits (i.e., if digits $d + 1, \dots, \text{numDigits} - 1$ were ignored). This holds for $d = 0$ by the correctness of counting sort (as argued above). Now suppose that this invariant holds with respect to digits $0, \dots, d$. Consider when processing digit $d + 1$. By property (1) above, after the next phase the elements will be sorted with respect to digit $d + 1$. Furthermore by property (2) the pass is stable and by the inductive hypothesis it was previously sorted with respect to digits $0, \dots, d$, so it follows that the elements are sorted to digits $0, \dots, d + 1$. Thus when the for loop over the digits has completed, *to* references the elements in sorted order.

Finally from the correctness of *getSize* and *getPosition*, the elements are placed back into the original collection in sorted order. ■

11.4.7 Bucket Sort

In this section we describe bucket sort. Unlike the sorting algorithms described so far, bucket sort does not directly take a comparator. Instead, the total order over the elements is defined by the provider bucketizer and the comparator that is included within it. (See Section 5.4 for a discussion of the *Bucketizer* interface and a sample implementation.) One can thus treat each element as a base *b* number and the comparisons are defined relative to this.

The *bucketSort* method with an argument takes *bucketizer*, the bucketizer to use.

```
public void bucketSort(Bucketizer<? super E> bucketizer) {
    bucketSortImpl(bucketizer);
}
```

The *bucketSortImpl* method that takes *bucketizer*, the bucketizer to use, and sorts the array using bucket sort. The comparator defined within the bucketizer must be such that all elements in bucket *i* are less than the elements in bucket $i + 1$. Bucket sort is a stable sort that applies the following five steps.

1. Determine how many items are in each bucket.
2. Make the counts cumulative, so *count[i]* is the number of items in that bucket or an earlier bucket.
3. Group the elements by bucket within the auxiliary array *temp*.

4. Put the elements back into the original array.
5. Use insertion sort, a stable sort, to sort the elements within the buckets (in a single pass).

```

protected void bucketsortImpl(Bucketizer<? super E> bucketizer) {
    int numBuckets = bucketizer.getNumBuckets();           //number of buckets
    int[] count = new int[getSize()];                     // # elements in each bucket
    Object[] temp = new Object[getSize()];                 //auxiliary array
    for (int i = 0; i < getSize(); i++)                      //Step 1
        count[bucketizer.getBucket(read(i))]++;
    for (int i = 1; i < numBuckets; i++)                     //Step 2
        count[i] += count[i-1];
    for (int i = getSize() - 1; i ≥ 0; i--) {                //Step 3
        Object x = read(i);                                 //go backwards so stable
        temp[--count[bucketizer.getBucket((E) x)]] = x;
    }
    for (int i = 0; i < size; i++)                           //Step 4
        put(i, temp[i]);
    insertionsort();                                         //Step 5
}

```

Correctness Highlights: As in the correctness proof for radix sort, after the end of Step 2, $count[i]$ is the number of items in that bucket or an earlier bucket. By definition of the bucketizer, all elements in bucket $i - 1$ must precede all elements in bucket i . Thus, in the final sorted array the elements in bucket i must occupy positions, $count[i - 1]$ to $count[i] - 1$. It is easily seen that Step 3 puts all elements into bucket i into these positions in the same relative order that they appeared in the array. Finally, since insertion sort is stable, the final array will be sorted with equal elements maintaining the same relative order. ■

11.5 Selection and Median Finding

In this section we consider the problem of computing the element at **rank** r , which is defined as the element that would be at position r of the collection if it were sorted. For example, the rank 0 element is a minimum element, the rank $\lfloor n/2 \rfloor$ element is the median, and the rank $n - 1$ element is a maximum element.

The algorithm we present was developed by Hoare [82] and later led to his development of quicksort. It uses the idea of partitioning the collection around a randomly selected element, and then recursively applying the selection process on portion of the collection that would have the rank r element. This method has expected linear cost (as opposed to the expected $\Theta(n \log n)$ cost of randomized quicksort). While this algorithm does not sort the collection, it does create mutations and upon completion it is guaranteed that the rank r element will be at position r .

The *repositionElementByRank* method that takes a single argument of r , the rank of the desired element in the sorted collection. It returns the element at rank r when using the default comparator. It throws a *PositionOutOfBoundsException* when r is not a valid position.

```

public E repositionElementByRank(int r) {
    return (E) repositionElementByRank(r, Objects.DEFAULT_COMPARATOR);
}

```

The *repositionElementByRank* method that takes *r*, the rank of the desired element in the sorted array, and *comp*, the comparator to use, returns the element at rank *r* when using the given comparator. It throws a *noSuchElementException* when the collection is empty or *r* is not a valid position.

```
public E repositionElementByRank(int r, Comparator<? super E> comp) {
    if (isEmpty() || r < 0 || r ≥ getSize())
        throw new NoSuchElementException();
    version.increment(); //invalidate all markers for iteration
    return (E) repositionElementByRankImpl(0, getSize()-1, r, comp);
}
```

Correctness Highlights: Follows from that of *repositionElementByRankImpl*, and the fact that the initial call made to that method bounds the search at the outermost ends of the range, with *r* falling within that range, otherwise a position out of bounds exception is thrown. The *partition* method used within *repositionElementByRankImpl* will invalidate all active markers for iteration unless no mutations are made to the collection. ■

The recursive *repositionElementByRank* method takes *left*, the leftmost position within the subarray, *right*, the rightmost position with the subarray, *r*, the rank of the desired element, and *comp*, the comparator to use when comparing the elements. It requires $left \leq r \leq right$.

```
E repositionElementByRankImpl(int left, int right, int r, Comparator<? super E> comp) {
    swap(getMedianOfThree(left, right, comp), right);
    int mid = partition(left, right, comp); //invalidates markers if a swap occurs
    if (r == mid) //rank r element is in its place
        return get(r);
    else if (r < mid) //rank r in part left of mid
        return repositionElementByRankImpl(left, mid-1, r, comp);
    else //rank r in part right of mid
        return repositionElementByRankImpl(mid+1, right, r, comp);
}
```

Correctness Highlights: By the correctness of *getMedianOfThree* and *swap*, the median of the left, middle, and right elements is placed as the right most element. By the correctness of *partition* all elements from positions *left* to *mid-1* are at most as large as the element in position *mid* (which is the pivot element). Likewise, the elements in positions *mid+1* to *right* are greater than the element in position *mid*. This implies that if $r = mid$, the element at position *r* is in its correct position, so it is properly returned. We now consider the remaining two cases:

$r < mid$: The rank *r* element must be in the left subarray, which means that only the left subarray needs to be processed. Also observe that by the requirements on the parameters $left \leq r$. Since $r < mid$, it follows that $left \leq r \leq mid - 1$, thus satisfying the conditions for the recursive call.

$r > mid$: The rank *r* element must be in the right subarray, which means that only the right subarray needs to be processed. Also observe that by the requirements on the parameters $r \leq right$. Since $r > mid$, it follows that $mid + 1 \leq r \leq right$, thus satisfying the conditions for the recursive call.

Since the length of the subarray is reduced by at least one each recursive call with $left \leq r \leq right$, eventually the base condition $r = mid$ is reached. ■

The basic approach to analyzing the expected time complexity is similar to that of randomized quicksort. When partitioning around a random element the expected time complexity is linear [82].

11.6 Basic Marker Inner Class

`AbstractCollection<E>.AbstractLocator<E>` implements *Locator<E>*

↑ **BasicMarker** implements *Locator<E>*

We define two marker classes in `Array`. The first one, `BasicMarker`, does not implement the *PositionalCollectionLocator* methods, such as *getCurrentPosition* and *addAfter*, but only implements the basic functionality of a marker. The reason for this design is that some uses of `Array`, namely `SortedArray` (Chapter 30), need to provide a locator that does not expose the extra methods of a positional collection locator. The second `Array` locator, `Marker` extends the `BasicMarker` class to include *PositionalCollectionLocator* methods, and is the type of marker returned by the `Array` *iterator*, *iteratorAtEnd*, and *iteratorAt* methods.

Each basic marker has a single instance variable, *pos* that stores the current position for the marker. Recall that we use a position of -1 (held in the constant *FORE*) for *FORE* and position *size* for *AFT*. Since *size* can be reduced by removing the last element without invalidating the markers, it is possible that *pos* may hold a value larger than *size*. Whenever this occurs, the marker is still logically at *AFT*. These choices for the internal representation of *FORE* and *AFT* reduce the boundary conditions in the code, similar to using a sentinel head and tail in a linked list.

```
int pos;                                //position for the marker
```

We introduce one representation property for the `BasicMarker` class that must be maintained by all methods.

MARKERLOC: $pos \geq -1$, when $pos = -1$ the marker is at *FORE*, and when $pos \geq size$ the marker is at *AFT*.

The constructor takes *p*, the position to store within the marker. It throws an *IllegalArgumentException* when **MARKERLOC** would be violated.

```
public BasicMarker(int p) {
    if (p < FORE || p > size)
        throw new IllegalArgumentException();
    this.pos = p;
    updateVersion();
}                                     //initialize modification count
```

| **Correctness Highlights:** **MARKERLOC** is enforced by the conditional. ■

The locator provides an accessor, *inCollection*, that returns *true* if and only if the locator is at an element of the collection.

```
public boolean inCollection() {
    return (pos > FORE && pos < size);
}
```

The locator provides an accessor, *get*, that returns the element stored at the current locator position. It throws a *NoSuchElementException* when the locator is at *FORE* or *AFT*.


```

public E get() {
    if (!inCollection())
        throw new NoSuchElementException();
    return (E) a[getIndex(pos)];
}

```

Correctness Highlights: Follows from MARKERLOC, and the correctness of *inCollection* and the Array *getIndex* method. ■

The *advance* method moves the locator forward by one position. It returns *true* if and only if after the update the locator is still at a position in the collection. The user program can use this return value to recognize when a locator has reached AFT. It throws an *AtBoundaryException* when the locator is at AFT since there is no place to advance.

```

public boolean advance() throws ConcurrentModificationException {
    checkValidity(); //check if marker has been invalidated
    if (pos ≥ size) //by MarkerLoc, marker is at AFT
        throw new AtBoundaryException("Already after end.");
    pos++; //move to the next position
    return (pos != size); //true iff marker now at AFT
}

```

Correctness Highlights: By MARKERLOC the exception is correctly thrown. Otherwise, incrementing the position advances the locator as specified. Since the locator only advances when $pos < size$, after the update $pos \leq size$. By MARKERLOC, the new locator position is AFT only when $pos = size$. Thus the correct value is returned. Finally, *checkValidity* is used to ensure that the marker is still valid (i.e., no critical mutations have been performed through a means other than this marker). ■

The *retreat* method moves the locator to the previous position. It returns *true* if and only if after the update the locator is still at a valid position. The user program can use this return value to recognize when the locator has reached FORE. It throws an *AtBoundaryException* when the locator is at FORE since then there is no place to retreat.

```

public boolean retreat() throws ConcurrentModificationException {
    checkValidity(); //check if marker has been invalidated
    if (pos == FORE) //by MarkerLoc, marker is at FORE
        throw new AtBoundaryException("Already before front.");
    if (pos > size) //if marker at AFT
        pos = size; //make sure its value is size
    pos--; //move to the previous position
    return (pos != FORE);
}

```

Correctness Highlights: By MARKERLOC the exception is correctly thrown. Observe that if the last element was removed one or more times, it is possible that pos is greater than $size$. If this is the case, it is reset to $size$ which is logically the desired value for AFT. Then decrementing pos moves the locator back one position as specified. By MARKERLOC, the locator only retreats when $pos > FORE$. So after the update $pos \geq FORE$. Finally, by MARKERLOC, the correct

value is returned since the new locator position is *FORE* only when *pos = FORE*. Finally, *checkValidity* is used to ensure that the marker is still valid (i.e., no critical mutations have been performed through a means other than this marker). ■

The *hasNext* method returns *true* if there is some element after the current locator position.

```
public boolean hasNext() throws ConcurrentModificationException {
    checkValidity();                //check if marker has been invalidated
    return (pos < size-1);
}
```

Correctness Highlights: Follows directly from *MARKERLOC*. Also *checkValidity* is used to ensure that the marker is still valid (i.e., no critical mutations have been performed through a means other than this marker). ■

As discussed in Section 5.8, when the application program calls *remove* through a locator and subsequently advances, the locator should be at the element that was just after the element removed. Hence, when an element is removed, *pos* should be moved to the previous position. This is accomplished using *retreat*. The *remove* method removes the element at the locator and updates the locator to be at the element in the collection preceding the one deleted. It throws a *NoSuchElementException* when the locator is at *FORE* or *AFT*. Finally, *checkValidity* is used to ensure that the marker is still valid (i.e., no critical mutations have been performed through a means other than this marker).

```
public void remove() throws ConcurrentModificationException {
    if (!inCollection())
        throw new NoSuchElementException();
    removeRange(pos, pos);          //invalidates all active locators
    updateVersion();                //ensures that this locator is still valid
    retreat();
}
```

Correctness Highlights: By the correctness of *inCollection* the exception is properly thrown. The rest of the correctness follows from that of the Array *removeRange* method and the *retreat* method. Also *checkValidity* is used to ensure that the marker is still valid (i.e., no critical mutations have been performed through a means other than this marker). ■

11.7 Marker Inner Class

`AbstractCollection<E>.AbstractLocator<E>` implements *Locator<E>*

└ BasicMarker implements *Locator<E>*

└ **Marker** implements *PositionalCollectionLocator<E>*

The Marker inner class extends BasicMarker to implement the *PositionalCollectionLocator* methods. Instances of this class are returned by the Array *iterator*, *iteratorAtEnd*, and *iteratorAt* methods. If a class that extends Array does not want to provide the functionality of a positional collection locator, a basic marker should instead be used.

```

public Marker(int p) {
    super(p);
}

```

The *getCurrentPosition* method returns the user position of the marker.

```

public int getCurrentPosition() throws ConcurrentModificationException {
    if (!inCollection())
        throw new NoSuchElementException();
    return pos;
}

```

Correctness Highlights: The correctness follows from that of *inCollection* and *MARKERLOC*. ■

The *addAfter* method takes *value*, the element to be added. It adds *value* to the collection after the marker location. It returns *null* since *Array* is an untracked implementation, and throws a *RuntimeException* when the marker is at *AFT*.

```

public PositionalCollectionLocator<E> addAfter(E value)
    throws ConcurrentModificationException {
    checkValidity(); //check if locator is valid
    if (pos ≥ size) //marker is at AFT
        throw new RuntimeException("can't add past the end");
    PositionalCollectionLocator<E> result;
    result = addImpl(pos+1, value); //invalidates all active locators
    updateVersion(); //ensures that this locator is still valid
    return result;
}

```

Correctness Highlights: By *MARKERLOC*, the exception is properly thrown, and if the marker is not at *AFT* then it should be added at position *pos+1*. The rest of the correctness follows from the *Array addImpl* method. *checkValidity* is used to ensure that the marker is still valid (i.e., no critical mutations have been performed through a means other than this marker) before the operation, and *updateVersion* ensures that the marker remains valid after the operation. ■

11.8 Performance Analysis

The asymptotic time complexities of all public methods for the *Array* class are given in [Table 11.8](#), and the asymptotic time complexities for all of the public methods of the *Array Marker* class are given in [Table 11.9](#).

The internal *getPosition* and *getIndex* methods take constant time since both just return their argument value. Also, the methods *nextIndex* and *prevIndex* execute in constant time since they just perform an increment or decrement operation. Since the above methods run in constant time and the element at index *p* in the underlying Java array can be accessed in constant time, the internal methods *getPosition*, *read*, and the public methods *get* and *set* take constant time. Also, the *swap* method takes constant time since locating the array slots that correspond to the given positions

method	time complexity
<i>add(o)</i>	$O(1)$
<i>addLast(o)</i>	$O(1)$
<i>get(p)</i>	$O(1)$
<i>iterator()</i>	$O(1)$
<i>iteratorAt(p)</i>	$O(1)$
<i>iteratorAtEnd()</i>	$O(1)$
<i>removeLast()</i>	$O(1)$
<i>set(p,o)</i>	$O(1)$
<i>swap(p1,p2)</i>	$O(1)$
<i>add(p,o)</i>	$O(n - p + 1)$
<i>remove(p)</i>	$O(n - p + 1)$
<i>removeRange(p,q)</i>	$O(n - p + 1)$
constructor with capacity <i>x</i>	$O(x)$
<i>ensureCapacity(x)</i>	$O(x)$
<i>accept(v)</i>	$O(n)$
<i>addFirst(o)</i>	$O(n)$
<i>addAll(c)</i>	$O(n)$
<i>bucketSort()</i>	$O(n)$ expected
<i>clear()</i>	$O(n)$
<i>contains(o)</i>	$O(n)$
<i>getLocator(o)</i>	$O(n)$
<i>positionOf(o)</i>	$O(n)$
<i>remove(o)</i>	$O(n)$
<i>removeFirst()</i>	$O(n)$
<i>repositionElementByRank(r)</i>	$O(n)$ expected
<i>toString()</i>	$O(n)$
<i>trimToSize()</i>	$O(n)$
<i>radixSort()</i>	$O(d(n + b))$
<i>heapSort()</i>	$O(n \log n)$
<i>mergeSort()</i>	$O(n \log n)$
<i>quickSort()</i>	$O(n \log n)$ expected
<i>treeSort()</i>	$O(n \log n)$
<i>insertionSort()</i>	$O(n^2)$
<i>retainAll(c)</i>	$O(n(c + n))$

Table 11.8 Summary of the asymptotic time complexities for the `PostionalCollection` public methods when using the array data structure. For the sorting and selection algorithms, the variations that take additional parameters (e.g., a comparator) have the same time complexity as their no argument counterparts. For radix sort, d is the maximum number of digits in any element and b is the base for the digitizer. The stated time complexity for bucket sort assumes that a constant number of elements are expected to occur in each bucket.

locator method	time complexity
constructor	$O(1)$
<i>advance()</i>	$O(1)$
<i>get()</i>	$O(1)$
<i>getCurrentPosition()</i>	$O(1)$
<i>hasNext()</i>	$O(1)$
<i>next()</i>	$O(1)$
<i>retreat()</i>	$O(1)$
<i>set(o)</i>	$O(1)$
<i>addAfter(o)</i>	$O(n)$
<i>remove()</i>	$O(n)$

Table 11.9 Summary of the time complexities for the public PositionalLocator methods for the array marker.

takes constant time and then three constant time statements are executed to actually perform the swap. The *iterator* method and *iteratorAtEnd* methods also take constant time since the marker constructor just sets a single instance variable to either -1 or *size*.

Any method that requires locating a given object *o* in the collection must, in the worst case, test each element for equivalence to *o*. Thus, *getLocator(o)*, *contains(o)*, *positionOf(o)*, and *remove(o)*, take linear time, in the worst case. Similarly, since the *accept*, *clear*, *toString*, and *traverseForVisitor* methods must iterate through the collection spending constant time at each element, they take linear time[§]. The *resizeArray* method must allocate a new array and copy all *n* elements into it, so it takes linear time. Hence, *trimToSize* takes linear time. Since Java initializes all slots in an array to *null*, both the constructor and *ensureCapacity* have time complexity linear in the size of the array allocated.

We now analyze the time complexity of *add(p,o)* which is used by all methods that insert elements into the collection. In order to insert an element into the underlying Java array at index *p* (which is the underlying index for position *p*), the *n - p* elements *a[p],...,a[n-1]* must be moved forward which takes $O(n - p)$ time. Then in constant time the given value can be stored in *a[p]*. Thus the time complexity of *add(p,o)* is $O(n - p) + O(1) = O(n - p + 1)$. Since *addLast(o)* and *add(o)*, call the *add* method with *p = n*, their time complexities are both $O(1)$. In contrast, since *addFirst(o)* calls the *add* method with *p = 0*, its time complexity is $O(n)$.

Finally, we analyze the time complexity of *removeRange(p,q)* which is used for all of the methods that remove elements. It takes constant time to determine how many elements are being removed, and to compute the arguments for the calls to the *arrayCopy* and *fill* methods. The elements in positions $0, \dots, p - 1$ are not touched, but all of the elements in positions $p + 1, \dots, n - 1$ are touched, either assigned from another element or assigned to *null*, in constant time operations, so the asymptotic time complexity for *removeRange(p,q)* is $O(n - p + 1)$. The time complexity for the other public methods to remove elements can be obtained by plugging in the appropriate value for *p*. The method *removeLast* has constant time complexity since *p = n - 1*. However, *removeFirst* takes $O(n)$ time since *p = 0*. The cost to remove the single element at position *p* is $O(n - p + 1)$ since *removeRange(p,p)* is called.

The *retainAll* method iterates through the collection removing elements that do not occur in the provided collection. The provided implementation calls *contains* for the collection *c*, which takes

[§]The Stringbuffer resizes periodically. As discussed in Chapter 13 this can be done so that the amortized cost is constant and hence the worst-case time complexity for *toString* is linear.

$O(|c|)$ time, and *remove* for the collection on which this method is called, which takes $O(n)$. Thus the resulting time complexity is $O(n(|c| + n))$. The time complexity would be reduced to $O(n \cdot |c|)$ by first marking which elements to remove, and then removing them all in a single pass. This implementation would take $O(n \cdot |c|)$ time to mark the elements to remove, and then $O(n)$ time to remove them. However, it would require an auxiliary array of size n .

The analysis for each sorting algorithm is given just after the correctness highlights. In summary, heap sort, merge sort, and tree sort *heapsort*, *mergesort*, and have worst-case $O(n \log n)$ complexity. Randomized quicksort has expected $O(n \log n)$ time complexity with the expectation taken only over the randomization used within the algorithm. No assumptions at all are made about the input. The worst-case time complexity for quicksort is $O(n^2)$, however this occurs with very low probability. Bucket sort has linear expected time complexity under the strong assumption that a constant number of elements are expected to occur in each bucket. However, if all elements fall into a single bucket, then bucket sort has $O(n^2)$ time complexity. Radix sort has time complexity $O(d(n + b))$ where b is the base for the digitizer, and d is the maximum number of digits of any element in the collection. Finally, insertion sort has $O(n^2)$ time complexity, however it is very efficient if the collection is already nearly sorted. In particular, if each element is a constant number of positions away from its final sorted position, then insertion sort has linear time complexity.

The *repositionElementByRank* method, when each partition is made using, a random element in the subarray, has linear expected time complexity. As with quicksort, no assumptions are made about the input – the expectation is just with respect to the randomization that occurs within the algorithm.

We now analyze the methods for the *Marker* class. Clearly the constructor takes constant time. Any methods that either access an element or update an instance variable (i.e., *get*, *set*, *getCurrentPosition*) run in constant time. Also, the methods *advance*, *retreat*, *hasNext*, and *next* are easily seen to run in constant time. In the worst-case, *addAfter* is called when the locator is at *FORE*, in which case all elements move forward so it takes linear time. Similarly, the time complexity of *remove* depends on the position of the element being removed. In the worst case, when the element in position 0 is removed, it takes linear time.

11.9 Further Reading

Sorting is a very well-studied problem that dates back to the earliest days of computers. Many of these earlier algorithms were passed on as “folk lore” making it hard to cite a paper that first defined some of the basic sorting algorithms. Knuth [97] and Mehlhorn [112] both provide comprehensive presentations of sorting and searching algorithms.

According to Knuth, a mechanical collator for merging decks of punched cards was invented in 1938. J. von Neumann wrote a program for merge sort on the EDVAC computer in 1945. By Knuth’s account, H.H. Seward invented counting sort in 1954, and also developed the idea of combining counting sort with radix sort. Radix sorting (beginning at the least significant digit) is credited to operators of mechanical card-sorting machines. The first published reference to radix sort is a 1929 document by L.J. Comrie. For a discussion of selecting the best variation of radix sort for practical use, see the article by McIlroy, et al. [111]. Bucket sort was first presented by E.J. Isaac and R.C. Singleton in 1956. Both heap sort [157] and tree sort [55] were developed in 1964.

The algorithm for selection we present in Section 15.6 is based upon the work of Hoare [82] who developed the idea of partitioning an array around a randomly element and then recursively applying the selection process on the appropriate portion of the array. Hoare then extended this approach to develop the quicksort algorithm [83]. Many variations of Hoare’s original partition algorithm have

been developed, such as the partition method by Nico Lomuto which is described by Bentley [23]. The median-of-three partition and other optimizations for quicksort are discussed by Bentley and McIlroy [24] and Sedgewick [134].

Blum et al. [26] presented a worst-case linear-time median finding algorithm. However, in practice, the selection algorithm of Hoare presented here is more efficient. Floyd and Rivest [57] present a variation of Hoare's algorithm in which the partition element is recursively selected from a small sample of elements.

See Cormen et al. [42] for a discussion of a variety of specialized sorting algorithms that apply under restricted conditions, and also for a presentation of the $\Omega(n \log n)$ lower bound proven by Ford and Johnson [59] for comparison-based sorting algorithms.

11.10 Quick Method Reference

Array Public Methods

- p. 129** *Array()*
- p. 129** *Array(int capacity)*
- p. 128** *Array(int capacity, Comparator<? super E> equivalenceTester)*
- p. 98** void **accept**(Visitor<? super E> v)
- p. 135** void **add**(E value)
- p. 135** void **add**(int p, E value)
- p. 100** void **addAll**(Collection<? extends E> c)
- p. 122** void **addFirst**(E value)
- p. 122** void **addLast**(E value)
- p. 157** void **bucketSort**(Bucketizer<? super E> bucketizer)
- p. 100** void **clear**()
- p. 97** boolean **contains**(E value)
- p. 99** void **ensureCapacity**(int capacity)
- p. 129** E **get**(int p)
- p. 96** int **getCapacity**()
- p. 96** Comparator<? super E> **getComparator**()
- p. 98** E **getEquivalentElement**(E target)
- p. 139** PositionalCollectionLocator<E> **getLocator**(E value)
- p. 96** int **getSize**()
- p. 148** void **heapSort**()
- p. 148** void **heapSort**(Comparator<? super E> comp)
- p. 143** void **insertionSort**()
- p. 143** void **insertionSort**(Comparator<? super E> comp)
- p. 96** boolean **isEmpty**()
- p. 139** PositionalCollectionLocator<E> **iterator**()
- p. 139** PositionalCollectionLocator<E> **iteratorAt**(int pos)
- p. 139** PositionalCollectionLocator<E> **iteratorAtEnd**()
- p. 145** void **mergesort**()
- p. 146** void **mergesort**(Comparator<? super E> comp)
- p. 131** int **positionOf**(E value)
- p. 150** void **quicksort**()
- p. 150** void **quicksort**(Comparator<? super E> comp)
- p. 155** void **radixSort**(Digitizer<? super E> digitizer)
- p. 138** boolean **remove**(E value)
- p. 137** E **remove**(int p)
- p. 137** E **removeFirst**()

- p. 137 E **removeLast**()
- p. 136 void **removeRange**(int fromPos, int toPos)
- p. 158 E **repositionElementByRank**(int r)
- p. 159 E **repositionElementByRank**(int r, Comparator<? super E> comp)
- p. 100 void **retainAll**(Collection<E> c)
- p. 133 E **set**(int p, E value)
- p. 133 void **swap**(int pos1, int pos2)
- p. 97 Object[] **toArray**()
- p. 97 E[] **toArray**(E[] array)
- p. 98 String **toString**()
- p. 138 void **traverseForVisitor**(Visitor<? super E> v)
- p. 149 void **treesort**()
- p. 149 void **treesort**(Comparator<? super E> comp)
- p. 99 void **trimToSize**()

Array Internal Methods

- p. 135 PositionalCollectionLocator<E> **addImpl**(int p, Object value)
- p. 158 void **bucketSortImpl**(Bucketizer<? super E> bucketizer)
- p. 148 void **buildPriorityQueue**(PriorityQueue<Object> pq)
- p. 136 void **closeGap**(int fromPos, int toPos)
- p. 97 int **compare**(E e1, E e2)
- p. 134 void **createGap**(int p)
- p. 97 boolean **equivalent**(E e1, E e2)
- p. 131 int **findPosition**(E value)
- p. 130 int **getIndex**(int p)
- p. 151 int **getMedianOfThree**(int left, int right, Comparator<? super E> sorter)
- p. 130 int **getPosition**(int index)
- p. 148 void **heapSortImpl**(Comparator sorter)
- p. 143 void **insertionSortImpl**(Comparator sorter)
- p. 146 void **mergeSortImpl**(Comparator<? super E> sorter)
- p. 146 void **mergeSortImpl**(Object[] data, Object[] aux, int left, int right, Comparator<? super E> sorter)
- p. 134 void **move**(int fromPos, int toPos)
- p. 132 Object[] **moveElementsTo**(Object[] newArray)
- p. 130 int **nextIndex**(int index)
- p. 152 int **partition**(int left, int right, Comparator<? super E> sorter)
- p. 130 int **prevIndex**(int index)
- p. 134 void **put**(int p, Object value)
- p. 150 void **quickSortImpl**(int left, int right, Comparator<? super E> sorter)
- p. 156 void **radixSortImpl**(Digitizer<? super E> digitizer)
- p. 129 E **read**(int p)
- p. 159 E **repositionElementByRankImpl**(int left, int right, int r, Comparator<? super E> comp)
- p. 132 void **resizeArray**(int desiredCapacity)
- p. 133 void **swapImpl**(int pos1, int pos2)
- p. 99 void **traverseForVisitor**(Visitor<? super E> v)
- p. 149 void **treesortImpl**(Comparator<? super E> sorter)
- p. 98 void **writeElements**(StringBuilder s)

Array.BasicMarker Public Methods

- p. 160 *BasicMarker*(int p)
- p. 161 boolean **advance**()
- p. 160 E **get**()
- p. 162 boolean **hasNext**()
- p. 101 void **ignoreConcurrentModifications**(boolean ignore)
- p. 101 void **ignorePriorConcurrentModifications**()
- p. 160 boolean **inCollection**()

- p. 101 E **next()**
- p. 162** void **remove()**
- p. 161** boolean **retreat()**

Array.BasicMarker Internal Methods

- p. 101 void **checkValidity()**
- p. 101 void **updateVersion()**

Array.Marker Public Methods

- p. 162** *Marker*(int p)
- p. 163** PositionalCollectionLocator<E> **addAfter**(E value)
- p. 163** int **getCurrentPosition()**

Chapter 12

Circular Array Data Structure

package `collection.positional`

`AbstractCollection<E>` implements *Collection<E>*

└ `Array<E>` implements *PositionalCollection<E>*

└ `CircularArray<E>` implements *PositionalCollection<E>*

Uses: Java primitive array

Used By: `Buffer` (Chapter 17), `AbstractGraph` (Chapter 53), `InTree` (Section 53.3)

Strengths: It has all of the advantages of an array (including random access by position and access and mutations at the back in constant time), as well as support for adding or removing an element from the front of the collection in constant time.

Weaknesses: In order to enable elements to efficiently be added to or removed from the front of the collection, there is computation required to convert between the user position and underlying index, making the accessors slightly less efficient than an array. As with `Array`, the user is responsible for calling a method to ensure sufficient capacity at appropriate times. Also, it takes linear time to add or remove an element for the middle portion of the array even if a locator is already positioned at the element.

Critical Mutators: *add* (except when $p = \text{size}$), *addFirst*, *addAll*, *clear*, *heapsort*, *insertionsort*, *mergesort*, *quicksort*, *radixsort*, *remove*, *removeFirst*, *removeRange* (except when $\text{toPosition} = \text{size} - 1$), *repositionElementByRank*, *retainAll*, *swap*, *treesort*

Competing Data Structures: If all elements are added or removed at the back of the collection, then an array is more efficient than a circular array. If constant time access via position is not required and elements are to be added to or removed from the middle portion of the array (via a locator), then a doubly linked list is a better option. Finally, if the properties of a circular array are acceptable, but a tracker is needed to keep track of the locations of the elements as they move, then a tracked circular array should be used instead.

12.1 Internal Representation

Each circular array uses the Java primitive array inherited from `Array`, and also introduces *start*, which identifies the underlying array index for the element at position 0. The name “circular array” is used since the underlying Java array is viewed as a circle, where $a[a.length-1]$ is followed by $a[0]$. In other words, the elements in the collection may wrap from the back of a to the front of a .

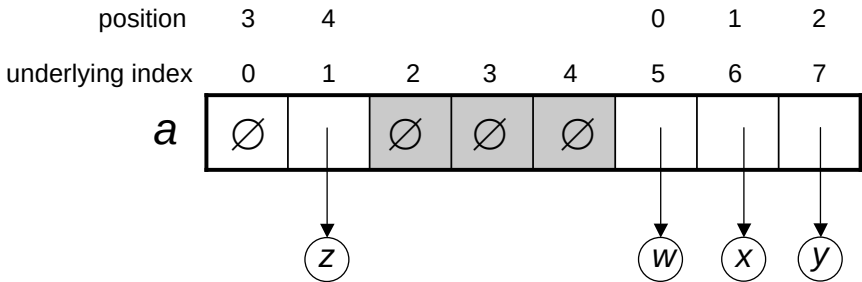


Figure 12.1
A circular array representing the positional collection $\langle w, x, y, null, z \rangle$ where $start=5$.

Mathematically, the Java modulus function (%) can be used to compute the underlying index that corresponds to a position of p as $(start+p) \% a.length$.

Allowing the user position and underlying array index to differ makes the data structure more complicated. However, this flexibility is what enables the circular array to efficiently add or remove elements from either end of the collection. As with Array, the capacity of the underlying array is always at least as large as the number of elements held in the collection. An exception is thrown if there is an attempt to add an element to the collection when the underlying array is full.

Instance Variables and Constants: All instance variables from *Array* are inherited, and we add an instance variable to keep track of the index of the underlying array that holds the element with a user index of 0.

`protected int start;`

`//index where the collection starts`

Populated Example: Figure 12.1 shows the internal representation for a *CircularArray* containing the position collection $\langle w, x, y, null, z \rangle$ of size 5. The capacity is 8 and $start = 5$. The elements of *a* that are not in use are shown as gray. As illustrated by position 3 in this collection, it is permissible for an element of the collection to be a *null*. The example of Figure 12.1 wraps. Figure 11.1 is another possible internal representation for the same circular array where $start=0$. In that case, the underlying Java array does not wrap.

Abstraction Function: The abstraction function for *CircularArray* differs from that of *Array*. Let *CA* be a circular array. The abstraction function

$$AF(CA) = \langle u_0, u_1, \dots, u_{size-1} \rangle \text{ such that } u_p = a[(start + p) \% a.length].$$

Observe that this abstraction function implies that the slots of the array in use are either:

- (a) a contiguous portion $a[start], a[start+1], \dots, a[start+size-1]$, or
- (b) the contiguous portion $a[start], a[start+1], \dots, a[a.length-1]$ and the contiguous portion $a[0], \dots, a[start + size - a.length - 1]$.

Terminology: We use the same terminology for *in use* and *valid position* as in our discussion of the *Array* data structure. We add the following definition that is specific to a circular array.

- When the array is of form (b) above, we say that circular array **wraps**.

12.2 Representation Properties

The representation properties maintained by `CircularArray` are the same as for `Array`, except `PLACEMENT` is overridden by:

PLACEMENT: Element u_p can be found using $start$ as an offset into the array, wrapping as needed. More formally, $a[(start + p) \% a.length] = u_p$ for all $p \in \{0, \dots, size-1\}$.

12.3 Methods

Generally, the methods from `Array` that are overridden here are those that relate to the translation between the user position and the index of the underlying array. Also, methods that move portions of the underlying array are overridden since the Java `System.arraycopy` cannot, in a single call, move a portion of an array that wraps.

12.3.1 Constructors

The most general constructor takes *capacity*, the desired initial capacity for the underlying array, and *equivalenceTester*, a user-provided equivalence tester, creates a circular array with the given capacity that uses the provided equivalence tester. It throws an *IllegalArgumentException* when *capacity* < 0 .

```
public CircularArray(int capacity, Comparator<? super E> equivalenceTester){
    super(capacity, equivalenceTester);
    start = 0;
}
```

Correctness Highlights: Follows from the correctness of the `Array` constructor since `PLACEMENT` is unaffected when *start* = 0. ■

Several additional convenience constructors are provided to replace some parameters by the default values. Their correctness follows from that of the above constructor. The constructor with no arguments creates a circular array with a default initial capacity that uses the default equivalence tester.

```
public CircularArray(){
    this(DEFAULT_CAPACITY, Objects.DEFAULT_EQUIVALENCE_TESTER);
}
```

The constructor that takes *capacity*, the desired initial capacity, creates a circular array with the given capacity that uses the default equivalence tester. It throws an *IllegalArgumentException* when *capacity* < 0 .

```
public CircularArray(int capacity) {
    this(capacity, Objects.DEFAULT_EQUIVALENCE_TESTER);
}
```

12.3.2 Representation Accessors

All representation accessors are modified to be consistent with the $u_p = a[(start + p \% a.length)]$. Recall that, the *getIndex* method takes p , a valid user position, and returns the corresponding index in the underlying array a .

```
protected final int getIndex(int p){
    int loc = start + p;
    if (loc < a.length)
        return loc;
    else
        return loc - a.length;
}
```

Correctness Highlights: By PLACEMENT, the value to return is $(start + position) \% a.length$. To improve efficiency we compute this value directly (without using Java's modulus operator). ■

Recall that *getPosition* takes *index*, an index of a that is in use, and returns the corresponding user position. According to the abstraction function, the correct position to return is $(index - start) \bmod a.length$. For example, in the circular array of Figure 12.1, *getPosition*(0) should return $(0 - 5) \% 8 = -5 \% 8 = 3$. This method requires that *index* is a valid position.

```
protected final int getPosition(int index){
    int pos = index - start;
    if (pos ≥ 0)
        return pos;
    else
        return pos + a.length;
}
```

Correctness Highlights: For the sake of efficiency, we directly perform the needed computation. The correctness follows from PLACEMENT. ■

Recall that *nextIndex* takes *index*, an underlying index of a , and returns the underlying index for the next element in the collection. The next index is computed without regard to the contents of that slot. If the parameter is the index of the last element of the collection, then *nextIndex* returns the index of the slot immediately after the end of the collection (wrapping if required), unless the *CircularArray* is at capacity, in which the index returned is that of the first element in the collection.

```
protected final int nextIndex(int index){
    if (index < a.length - 1)
        return index+1;
    else
        return 0;
}
```

Correctness Highlights: By PLACEMENT, unless *index* is at the last array slot (i.e., $a.length - 1$), the next slot is at $index+1$. If the index is at the last array slot, then by PLACEMENT the next index is 0. ■

Similarly, *prevIndex* takes *index*, an underlying index, and returns the underlying index for the previous element in the collection. If the parameter is the index of the first element of the collection, then *prevIndex* returns the index of the slot immediately before the start of the collection (wrapping if required), unless the *CircularArray* is at capacity, in which case the index returned is that of the last element in the collection.

```
protected final int prevIndex(int index){
    if (index > 0)
        return index-1;
    else
        return a.length-1;
}
```

Correctness Highlights: Similar to *nextIndex*, we return the prior index, unless the parameter is 0, in which case we wrap back to the last array slot (index of $a.length - 1$). Again the correctness directly follows from PLACEMENT. ■

12.3.3 Representation Mutators

Recall that the internal method *resizeArray* method calls the method *moveElementsTo* to copy the elements from the old array to the new one. It takes *newArray*, the array to which the elements should be moved, and returns the parameter value for convenience. We need to override the method here since the elements may wrap. For simplicity, we copy into the new array beginning at slot 0. Therefore, *start* is reset to 0 at the end of the method. Since Java's *System.arraycopy* can only move a contiguous portion of the array when it does not wrap, when the array wraps it must be moved as two pieces.

```
protected Object[] moveElementsTo(Object[] newArray) {
    if (a != null) {
        if (start+size ≤ a.length)                                //a doesn't wrap
            System.arraycopy(a, start, newArray, 0, size);
        else{                                                    //a wraps, so copy in two sections
            System.arraycopy(a, start, newArray, 0, a.length-start);
            System.arraycopy(a, 0, newArray, a.length-start, size-a.length+start);
        }
    }
    start = 0;
    return newArray;
}
```

Correctness Highlights: We now argue that *moveElementsTo* maintains the four representation properties. SIZE is not affected since neither *size* nor *n* change. CAPACITY is preserved since resizing is prevented by the *resize* method if *desiredCapacity* < *size* (in which case an exception is properly thrown). Upon entering the method, PLACEMENT guarantees that for all $p \in \{0, 1, \dots, size-1\}$, $a[p] = u_{(p+start) \% a.length}$. When the elements are copied into the new array, they are placed beginning at index 0 and *start* is set to 0 to reflect this. It is easily verified that upon completion the elements of the collection are placed contiguously in $b[0], \dots, b[size-1]$.

That is, $b[p] = u_p$ for all $p \in \{0, 1, \dots, \text{size}-1\}$, so we maintain PLACEMENT with $\text{start} = 0$. Finally, NONRETENTION is preserved since it is assumed that the provided array is a fresh array, with all slots initialized to *null* by Java when the array is allocated. Since all elements remain in the same positions, the active markers can continue to be used for iteration. ■

The methods *ensureCapacity* and *trimToSize* are inherited.

12.3.4 Content Mutators

We first introduce two internal methods that shift a portion of the underlying array to the right or to the left as needed either to make space for inserting an element, or to remove the gap left by elements that have been removed.

We first describe the internal method *shiftLeft* that takes $p1$, the starting position to shift, $p2$, the ending position for the shift, and num , the number of slots to shift. It shifts the elements held in $p1, \dots, p2$ (possibly wrapped) left num slots. It will overwrite the num positions left of $p1$. It requires that $p1 \leq p2$, and $0 < \text{num} \leq a.\text{length} - (p2 - p1 + 1)$. The amount of the shift is at least 1 and is not so much that the portion of the array being shifted would intersect with itself.

This method must handle both the cases when the portion of the underlying array holding positions $p1$ to $p2$ does and does not wrap. The cases in the algorithm are illustrated in [Figures 12.2](#) and [12.3](#), where i is the underlying index for the element at position $p1$, and j is the underlying index for the element at position $p2$. Note that the cases depend only on whether the portion of the underlying array being shifted wraps, so case 1 (not wrapping) may apply even if the collection as a whole wraps. We define the **jeopardized** region as the num slots immediately left of slot i (in the wrapped array). The correctness arguments of the methods that use *shiftLeft* require that it does not change the value of any slots except those stored in the jeopardized region.

```
protected void shiftLeft(int p1, int p2, int num){
    int numToMove = p2 - p1 + 1;
    int i = getIndex(p1);                //underlying index for p1
    int j = getIndex(p2);                //underlying index for p2
    int cap = a.length;                  //capacity of a
    if (i ≤ j){                          //Case 1: a[i]...a[j] doesn't wrap
        if (i ≥ num)                     //Case 1a: a[i]...a[j] moves left, won't wrap
            System.arraycopy(a, i, a, i-num, numToMove);
        else if (num ≥ j+1)              //Case 1b: shifts past array boundary, won't wrap
            System.arraycopy(a, i, a, i-num+cap, numToMove);
        else {                           //Case 1c: i < num < j+1, a[i], ...a[j] will now wrap
            System.arraycopy(a, i, a, i-num+cap, num-i);           //block 1
            System.arraycopy(a, num, a, 0, numToMove-(num-i));      //block 2
        }
    } else {                             //Case 2: a[i]...a[j] does wrap
        System.arraycopy(a, i, a, i-num, cap-i);                  //block 1 (2a and 2b)
        if (num > j)                                               //Case 2a: all of front wraps to back
            System.arraycopy(a, 0, a, cap-num, j+1);              //block 2
        else {                                                     //Case 2b: portion of front wraps to back
            System.arraycopy(a, 0, a, cap-num, num);              //block 2
            System.arraycopy(a, num, a, 0, j+1-num);              //block 3
        }
    }
}
```

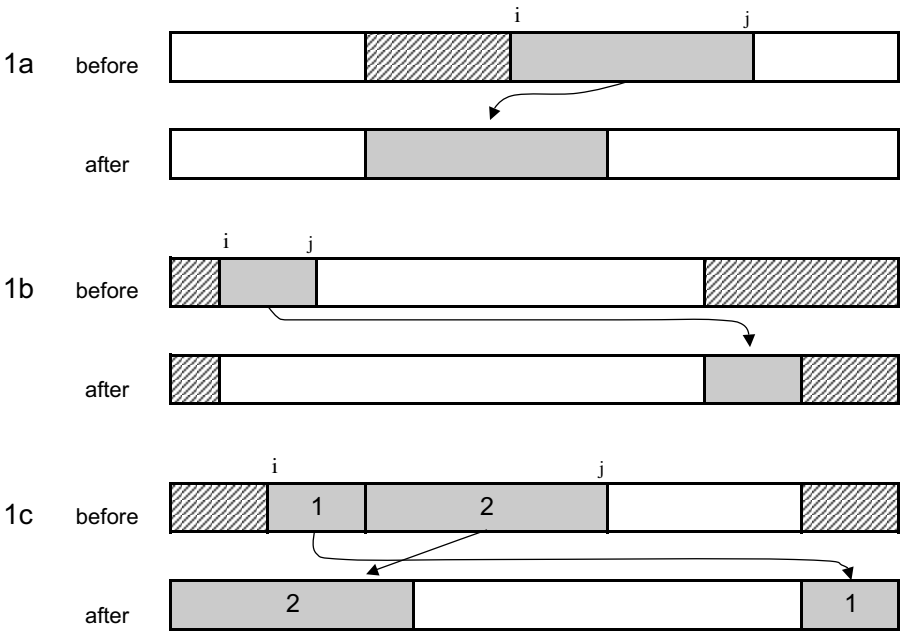


Figure 12.2
Case 1 for the *shiftLeft* procedure occurs if $a[i] \dots a[j]$ does not wrap prior to being shifted. The jeopardized region is shown in a diagonal fill pattern, and $a[i] \dots a[j]$ is filled in gray. Note that in cases 1a and 1c, if the jeopardized region is larger than the region being shifted (not pictured), then some of the jeopardized region will remain as illustrated in case 1b. Case 1b, can only occur when the jeopardized region is larger than the region being shifted.

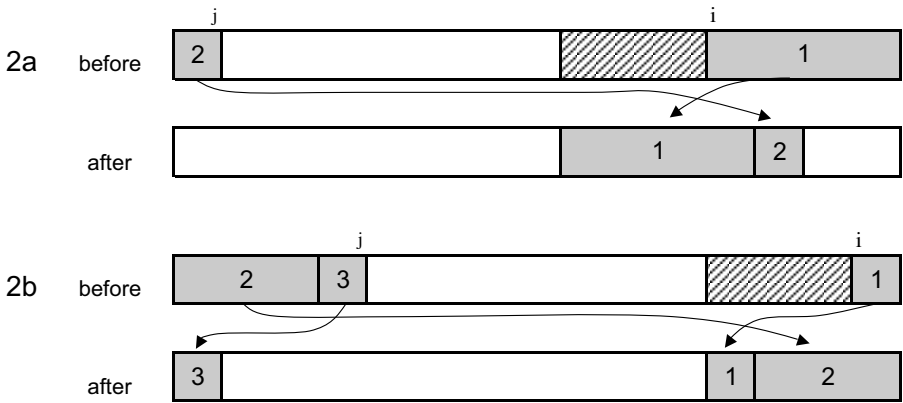


Figure 12.3
Case 2 for the *shiftLeft* procedure which occurs if $a[i] \dots a[j]$ wraps prior to being shifted. The shading follows the same convention as Figure 12.2. Case 2b can only occur when the jeopardized region is smaller than the region being shifted.

Correctness Highlights: We argue that *shiftLeft* behaves according to its specification as long as $p1 \leq p2$ and $0 < num < a.length - (p2 - p1 + 1)$.

We first consider Case 1 which is illustrated in [Figure 12.2](#). Cases 1a and 1b are easily verified to behave according to the specification. For Case 1c, some care is required to ensure that the first call to *arraycopy* only involves slots that are either in the jeopardized region, or those being copied. In other words, we must ensure that no elements in block 2 are overwritten by the first call to *arraycopy*. Observe that $p2 - p1 + 1$ is the number of elements being moved. It is required that *num* is at most the capacity of the underlying array minus the number of elements being moved, so the first blocked moved in case 1c cannot overlap the second block moved, so the correctness follows.

Next we consider Case 2 which is illustrated in [Figure 12.3](#). As in Case 1, what needs to be proven is that whenever a block is moved with *arraycopy*, any slots that are overwritten must fall into one of these categories: (1) slots in the jeopardized region, (2) slots in the region of elements to be moved that have already been moved in an earlier call to *arraycopy*, or (3) slots of the block being copied in the current call to *arraycopy*.

We first consider the *arraycopy* call that shifts block 1 (which is done for both Cases 2a and 2b). As with Case 1c, since the number of slots the elements are being shifted is at most the size of the collection minus the number of elements being moved, the first blocked moved in Case 2 cannot overlap any of the other blocks that are going to be moved. The block being moved only overlaps itself and slots in the jeopardized region. In Case 2a, the only block left to move is block 2, which by construction only overlaps portions that were originally in block 1 (but are not in the new location for block 1), the jeopardized region, or portions of block 2.

Finally, we consider Case 2b. The new location for block 2 only overlaps the portion of the array that originally held the right portion of block 1, but which does not overlap the new location of block 1. Finally, the new location of block 3 only overlaps the portion that used to hold block 2 but do not overlap the new location of block 2. Thus Case 2 also behaves as specified. ■

We have a similar method *shiftRight* that takes *p1*, a valid position, *p2*, a valid position, and *num*, the number of positions to shift. It shifts the elements held in *p1*, ..., *p2* (possibly wrapped) right *num* slots. It will overwrite the *num* positions right of *p2*. This method requires that $p1 \leq p2$, and $0 < num \leq a.length - (p2 - p1 + 1)$. The amount to shift is at least 1, and the portion of the array being shifted does intersect with itself. This method works in a symmetric manner to *shiftLeft*.

```
protected void shiftRight(int p1, int p2, int num){
    int numToMove = p2 - p1 + 1;
    int i = getIndex(p1);
    int j = getIndex(p2);
    int cap = a.length;
    if (i ≤ j){
        //a[i]...a[j] doesn't wrap
        if (num ≤ cap-j-1) //a[i]...a[j] moves right, doesn't wrap
            System.arraycopy(a, i, a, i+num, numToMove);
        else if (num ≥ cap-i) //moves right past array boundary
            System.arraycopy(a, i, a, i+num-cap, numToMove);
        else { //n-j+1 < num < n-i, a[i]...a[j] will wrap
            System.arraycopy(a, cap-num, a, 0, numToMove-cap+num+i);
            System.arraycopy(a, i, a, num+i, cap-(num+i));
        }
    } else {
        //a[i]...a[j] does wrap
```

```

    System.arraycopy(a, 0, a, num, j+1);           //shift a[0]..a[j] right num slots
    if (num ≥ cap-i)                               //all of end wraps to front
        System.arraycopy(a, i, a, 0, num);
    else {                                         //only portion of end wraps to front
        System.arraycopy(a, cap-num+1, a, 0, num);
        System.arraycopy(a, i, a, i+num, cap-i-num);
    }
}

```

Correctness Highlights: It is analogous to *shiftLeft*. ■

Methods to Perform Insertion

All of the inherited methods that insert elements do so by making an appropriate call to the *add* method that in turns calls *createGap* to move the required elements. This method is overridden from *Array* to improve efficiency by either moving the $p + 1$ elements $u_0, \dots, u_p$ left, or the $size - p$ elements $u_p, \dots, u_{size-1}$ right, in order to minimize the number of elements moved. Recall *createGap* takes p , a valid user position where the gap is to be created, and moves elements in such a way to increment the position number for the elements that were at positions $p, \dots, size - 1$ to be at positions $p+1, \dots, size$.

```

protected void createGap(int p) {
    if (p + 1 ≤ size - p) {                       //if # to move left ≤ # to move right
        if (p != 0)                               //and not adding to front
            shiftLeft(0, p-1, 1);                 //move front portion left
        start = prevIndex(start);                 //back up start by one
    }
    else if (p != size)                           //if not adding to back
        shiftRight(p, size-1, 1);                 //move end portion right
}

```

Correctness Highlights: By the correctness of *prevIndex*, when p is 0, the specification of this method is satisfied by moving *start* back by one.

Next we argue that the conditions required by *shiftLeft* and *shiftRight* are satisfied, so the correctness of those methods will follow. In both cases, $num = 1$ which satisfies $num > 0$, and it is easily checked that $p1 \leq p2$. We now argue that in both cases $num \leq a.length - (p2 - p1 + 1)$. For the call to *shiftLeft*, $p1$ is 0 and $p2$ is $p - 1$, so $a.length - (p2 - p1 + 1) = a.length - p$. Combined with the observation that *shiftLeft* is never called when $p = size - 1$ and that $size \leq a.length$ yields the needed requirement that

$$num = 1 \leq size - p \leq a.length - p.$$

For the call to *shiftRight*, $p1 = p$ and $p2 = size - 1$, yielding that $size - (p2 - p1 + 1) = p$. Since *shiftRight* is not called when $p = 0$, and $size \leq a.length$, it follows that $num = 1 \leq p \leq a.length - (p2 - p1 + 1)$. Thus, the shift methods will correctly perform the specified shift. The rest of the correctness thus follows from that of *shiftLeft* and *shiftRight*.

Observe that if $p = 0$ or $p = size$ then no elements are shifted which leaves all active markers valid. In all other cases, either the call to *shiftLeft* or call to *shiftRight* invalidates the active markers. ■

Methods to Perform Deletion

We now consider the changes needed to remove elements from the collection. First, we override *setToNull* to address the case in which the array wraps. Recall that *setToNull* takes *pos1*, a valid position, and *pos2*, a valid position. This method sets positions *pos1*, $\dots$, *pos2* to *null*. This method requires that $pos1 \leq pos2$.

```
protected void setToNull(int pos1, int pos2){
    int i = getIndex(pos1);                //index for pos1
    int j = getIndex(pos2);                //index for pos2
    if (i ≤ j)                             //circular array does not wrap
        Arrays.fill(a, i, j+1, null);
    else {                                  //circular array wraps
        Arrays.fill(a, i, a.length, null);
        Arrays.fill(a, 0, j+1, null);
    }
}
```

All of the inherited methods that remove elements do so by making an appropriate call to the *removeRange* method that in turns calls *closeGap*. This method is overridden from *Array* to improve efficiency by minimizing the number of elements that must be moved. Recall that *closeGap* takes *fromPos*, the first position in the gap to close, and *toPos*, the last position in the gap to close. This method moves elements to decrement the position number for elements that were at positions $toPos + 1, \dots, size - 1$ to positions $fromPos, \dots, size - 1 - (toPos - fromPos + 1)$. Furthermore, the indices in the underlying array that are no longer in use are set to *null*.

```
protected void closeGap(int fromPos, int toPos){
    int sizeOfFrontPortion = fromPos;      //introduced for clarity
    int sizeOfBackPortion = size-toPos-1;  //also introduced for clarity
    int numRemoving = toPos - fromPos + 1; //number of elements removed
    if (sizeOfBackPortion ≤ sizeOfFrontPortion){ //shift back portion left
        if (toPos != size-1)
            shiftLeft(toPos+1, size-1, numRemoving);
        setToNull(size-numRemoving, size-1);
    } else {                                //shift front portion right
        if (fromPos != 0)
            shiftRight(0, fromPos-1, numRemoving);
        setToNull(0, numRemoving-1);
        start = getIndex(numRemoving);     //i.e., (start+numRemoving)%a.length
    }
}
```

Correctness Highlights: Either the elements in the back portion (positions $toPos + 1, \dots, size - 1$) must be moved to the front to close the gap (if there is one), or the front portion (positions $0, \dots, fromPos - 1$) must be moved toward the back to close the gap (if there is one). This method selects the option that moves the fewest elements since the time complexity is dominated by the cost of moving these elements.

We first consider when $sizeOfBackPortion \leq sizeOfFrontPortion$. We argue that if *shiftLeft* is called, its requirements are met. Plugging in the parameter values gives that the requirements are

$toPos < size - 1$ and $0 < numRemoving \leq a.length + toPos + 1 - size$. The first of these follows since $toPos$ is a valid position not equal to $size - 1$. By the requirement that $fromPos \leq toPos$, $numRemoving \geq 0$. The final requirement is needed to ensure that the portion of the array being shifted does not overlap itself. Algebraically, this condition becomes $a.length \geq size - fromPos$, which holds by CAPACITY. Thus *shiftLeft* is guaranteed to perform as specified, which will in turn guarantee the elements that were at positions $toPos + 1, \dots, size - 1$ will end at positions $fromPos, \dots, size - 1 - (toPos - fromPos + 1)$. Finally, the call to *setToNull* satisfies the rest of the specification.

Next we consider when $sizeofBackPortion > sizeofFrontPortion$. We argue that if *shiftRight* is called, its requirements are met. Plugging in the parameter values gives that the requirements are $fromPos \geq 1$ and $0 < numRemoving \leq a.length - fromPos$. The first of these follows since $fromPos$ is a valid position not equal to 0. By the requirement that $fromPos \leq toPos$, $numRemoving \geq 0$. The final requirement clearly holds since the shifted portion does not overlap itself. Algebraically, this condition becomes $a.length \geq toPos + 1$ which holds from CAPACITY since $a.length \geq size$ and $size - 1 \geq toPos$. Thus *shiftRight* is guaranteed to perform as specified, so the elements in the collection remain contiguous in the circular array. After updating *start*, the elements that were at positions $toPos + 1, \dots, size - 1$ will end at positions $fromPos, \dots, size - 1 - (toPos - fromPos + 1)$ as specified. Finally, the call to *setToNull* satisfies the rest of the specification.

The active markers only need to be invalidated when elements in the array are shifted. When executed, both *shiftLeft* and *shiftRight* invalidate the active markers. ■

12.4 Performance Analysis

The asymptotic the time complexities for the positional collection methods of *CircularArray* are compared to those of *Array* in Table 12.4. Most of the analysis for *CircularArray* is the same as for *Array*. We only discuss the methods for which the time complexity is different. To insert an element at position p , either the p elements before position p must be moved back, or the $n - p$ elements from position p to the end must be moved forward. This takes $O(\min(p, n - p + 1))$ time. Then, in constant time the given value can be stored and the related variables can be updated. Thus the time complexity of *add(p,v)* is $O(\min(p + 1, n - p + 1))$.

To execute the method *removeRange(p,q)*, either the p elements before position p must be shifted right or the $n - q - 1$ elements after position q must be shifted left. In addition, it takes $O(q - p + 1)$ to set the removed elements to *null*. Thus, the overall asymptotic time complexity for *removeRange(p,q)* is $O(\min(q + 1, n - p + 1))$.

The time complexity for all the other methods that remove an element are computed by just replacing p and q by the appropriate parameter. Since *removeFirst* sets $p = q = 0$, it has constant time complexity. Since *removeLast* last sets $p = q = n - 1$, it also has constant time complexity. Finally, the time complexity for *remove(p)* is $O(\min(p + 1, n - p + 1))$ since *removeRange(p,p)* is called.

For the locator methods, the asymptotic time complexities are the same as for the *Array* as shown in Table 11.9. The key difference is that the time to add or remove is proportional to the minimum of the number of elements in the collection before or after the position where the new element is being added or removed, whereas for the array data structure it is always proportional to the number of elements after the position where the change is occurring. So in the worst-case, $n/2$ elements as opposed to n elements must be moved, but both have linear asymptotic time complexity.

method	circular array time complexity	array time complexity
<i>add(o)</i>	$O(1)$	$O(1)$
<i>addFirst(o)</i>	$O(1)$	$O(n)$
<i>addLast(o)</i>	$O(1)$	$O(1)$
<i>get(p)</i>	$O(1)$	$O(1)$
<i>iteratorAt(p)</i>	$O(1)$	$O(1)$
<i>removeFirst()</i>	$O(1)$	$O(n)$
<i>removeLast()</i>	$O(1)$	$O(1)$
<i>set(p,o)</i>	$O(1)$	$O(1)$
<i>swap(p,q)</i>	$O(1)$	$O(1)$
<i>add(p,o)</i>	$O(\min(p+1, n-p+1))$	$O(n-p+1)$
<i>remove(p)</i>	$O(\min(p+1, n-p+1))$	$O(n-p+1)$
<i>removeRange(p,q)</i>	$O(\min(q+1, n-p+1))$	$O(n-p+1)$
constructor with capacity x	x	$O(x)$
<i>ensureCapacity(x)</i>	$O(x)$	$O(x)$
<i>accept(v)</i>	$O(n)$	$O(n)$
<i>addAll(c)</i>	$O(n)$	$O(n)$
<i>bucketSort()</i>	$O(n)$ expected	$O(n)$ expected
<i>clear()</i>	$O(n)$	$O(n)$
<i>contains(o)</i>	$O(n)$	$O(n)$
<i>getLocator(o)</i>	$O(n)$	$O(n)$
<i>positionOf(o)</i>	$O(n)$	$O(n)$
<i>remove(o)</i>	$O(n)$	$O(n)$
<i>repositionElementByRank(r)</i>	$O(n)$ expected	$O(n)$ expected
<i>toString()</i>	$O(n)$	$O(n)$
<i>trimToSize()</i>	$O(n)$	$O(n)$
<i>radixSort()</i>	$O(d(n+b))$	$O(d(n+b))$
<i>heapSort()</i>	$O(n \log n)$	$O(n \log n)$
<i>mergeSort()</i>	$O(n \log n)$	$O(n \log n)$
<i>quickSort()</i>	$O(n \log n)$ expected	$O(n \log n)$ expected
<i>treeSort()</i>	$O(n \log n)$	$O(n \log n)$
<i>insertionSort()</i>	$O(n^2)$	$O(n^2)$
<i>retainAll(c)</i>	$O(n(c +n))$	$O(n(c +n))$

Table 12.4 Summary of the asymptotic time complexities for the `PostionalCollection` public methods when using a circular array, and for comparison an array, to implement it. For the sorting and selection algorithms, the variations that take additional parameters (e.g., a comparator) have the same time complexity as their no argument counterparts. For radix sort d is the maximum number of digits in any element, and b is the base for the digitizer. The stated time complexity for bucket sort assumed that a constant number of elements are expected to occur in each bucket.

12.5 Quick Method Reference

CircularArray Public Methods

- p. 173 *CircularArray*()
- p. 173 *CircularArray*(int capacity)
- p. 173 *CircularArray*(int capacity, Comparator<? super E> equivalenceTester)
- p. 98 void **accept**(Visitor<? super E> v)
- p. 135 void **add**(E value)
- p. 135 void **add**(int p, E value)
- p. 100 void **addAll**(Collection<? extends E> c)
- p. 122 void **addFirst**(E value)
- p. 122 void **addLast**(E value)
- p. 157 void **bucketSort**(Bucketizer<? super E> bucketizer)
- p. 100 void **clear**()
- p. 97 boolean **contains**(E value)
- p. 99 void **ensureCapacity**(int capacity)
- p. 129 E **get**(int p)
- p. 96 int **getCapacity**()
- p. 96 Comparator<? super E> **getComparator**()
- p. 98 E **getEquivalentElement** (E target)
- p. 139 PositionalCollectionLocator<E> **getLocator**(E value)
- p. 96 int **getSize**()
- p. 148 void **heapSort**()
- p. 148 void **heapSort**(Comparator<? super E> comp)
- p. 143 void **insertionSort**()
- p. 143 void **insertionSort**(Comparator<? super E> comp)
- p. 96 boolean **isEmpty**()
- p. 139 PositionalCollectionLocator<E> **iterator**()
- p. 139 PositionalCollectionLocator<E> **iteratorAt**(int pos)
- p. 139 PositionalCollectionLocator<E> **iteratorAtEnd**()
- p. 145 void **mergeSort**()
- p. 146 void **mergeSort**(Comparator<? super E> comp)
- p. 131 int **positionOf**(E value)
- p. 150 void **quickSort**()
- p. 150 void **quickSort**(Comparator<? super E> comp)
- p. 155 void **radixSort**(Digitizer<? super E> digitizer)
- p. 138 boolean **remove**(E value)
- p. 137 E **remove**(int p)
- p. 137 E **removeFirst**()
- p. 137 E **removeLast**()
- p. 136 void **removeRange**(int fromPos, int toPos)
- p. 158 E **repositionElementByRank**(int r)
- p. 159 E **repositionElementByRank**(int r, Comparator<? super E> comp)
- p. 100 void **retainAll**(Collection<E> c)
- p. 133 E **set**(int p, E value)
- p. 133 void **swap**(int pos1, int pos2)
- p. 97 Object[] **toArray**()
- p. 97 E[] **toArray**(E[] array)
- p. 98 String **toString**()
- p. 138 void **traverseForVisitor**(Visitor<? super E> v)
- p. 149 void **treeSort**()
- p. 149 void **treeSort**(Comparator<? super E> comp)
- p. 99 void **trimToSize**()

CircularArray Internal Methods

- p. 135 `PositionalCollectionLocator<E> addImpl(int p, Object value)`
- p. 158 `void bucketsortImpl(Bucketizer<? super E> bucketizer)`
- p. 148 `void buildPriorityQueue(PriorityQueue<Object> pq)`
- p. 136 `void closeGap(int fromPos, int toPos)`
- p. 97 `int compare(E e1, E e2)`
- p. 134 `void createGap(int p)`
- p. 97 `boolean equivalent(E e1, E e2)`
- p. 131 `int findPosition(E value)`
- p. 130 `int getIndex(int p)`
- p. 151 `int getMedianOfThree(int left, int right, Comparator<? super E> sorter)`
- p. 130 `int getPosition(int index)`
- p. 148 `void heapsortImpl(Comparator sorter)`
- p. 143 `void insertionsortImpl(Comparator sorter)`
- p. 146 `void mergesortImpl(Comparator<? super E> sorter)`
- p. 146 `void mergesortImpl(Object[] data, Object[] aux, int left, int right,
 Comparator<? super E> sorter)`
- p. 134 `void move(int fromPos, int toPos)`
- p. 132 `Object[] moveElementsTo(Object[] newArray)`
- p. 130 `int nextIndex(int index)`
- p. 152 `int partition(int left, int right, Comparator<? super E> sorter)`
- p. 130 `int prevIndex(int index)`
- p. 134 `void put(int p, Object value)`
- p. 150 `void quicksortImpl(int left, int right, Comparator<? super E> sorter)`
- p. 156 `void radixsortImpl(Digitizer<? super E> digitizer)`
- p. 129 `E read(int p)`
- p. 159 `E repositionElementByRankImpl(int left, int right, int r, Comparator<? super E> comp)`
- p. 132 `void resizeArray(int desiredCapacity)`
- p. 180** `void setToNull(int pos1, int pos2)`
- p. 176** `void shiftLeft(int p1, int p2, int num)`
- p. 178** `void shiftRight(int p1, int p2, int num)`
- p. 133 `void swapImpl(int pos1, int pos2)`
- p. 99 `void traverseForVisitor(Visitor<? super E> v)`
- p. 149 `void treesortImpl(Comparator<? super E> sorter)`
- p. 98 `void writeElements(StringBuilder s)`

Chapter 13

Dynamic Array and Dynamic Circular Array Data Structures

The dynamic array extends the array to automatically change the capacity as needed. The dynamic circular array is identical to the dynamic array except that it instead extends the circular array. Thus we just briefly discuss the dynamic circular array and do not provide the implementation in the book. The full implementation is on the CD included with this book.

13.1 Dynamic Array

package `collection.positional`

`AbstractCollection<E>` implements *Collection<E>*

↑ `Array<E>` implements *PositionalCollection<E>*

↑ **`DynamicArray<E>`** implements ***PositionalCollection<E>***

Uses: Java primitive array

Used By: Stack (Chapter 19), SortedArray (Chapter 30), BinaryHeap (Chapter 25), AbstractWeightedGraph (Chapter 57)

Strengths: Random access in constant time by position is supported and the array is automatically resized in such a way to guarantee constant amortized cost for the insert or remove operations associated with resizing. Thus, in amortized constant time an element can be added or removed at the back of the collection.

Weaknesses: There is a small amount of overhead required for the automatic resizing. Like the Array data structure, inserting or removing an element near the front of the collection has linear cost.

Critical Mutators: *add* (except when $p = \text{size}$), *addFirst*, *addAll*, *clear*, *heapsort*, *insertionsort*, *mergesort*, *quicksort*, *radixsort*, *remove*, *removeFirst*, *removeRange* (except when $\text{toPosition} = \text{size} - 1$), *repositionElementByRank*, *retainAll*, *swap*, *treesort*

Competing Data Structures: If the number of elements to be held by the positional collection is known at initialization, then resizing is not needed, in which case the array data structure is a better choice. If elements will be added or removed from the front of the collection then a dynamic circular array should be used. If there is a need to efficiently remove or add elements from the middle portion of the collection, or to avoid the cost of resizing, then a list-based implementation should be considered. Finally, if a tracker versus a marker is needed then consider a tracked array.

13.2 Internal Representation

The `DynamicArray` extends `Array` by modifying methods that add or remove elements to perform resizing as needed. Whenever resizing occurs, the elements must be copied to a fresh array. This takes time, so we do not wish to resize very often. Therefore, we want to leave extra space in the array to avoid resizing again in the near future. However, we do not want to waste too much space either. There is an inherent trade-off between the amount of extra space and how often it is necessary to resize. To balance time to resize with the amount of unused space, whenever the underlying array grows, its size is doubled. As discussed in Section 13.5, this approach guarantees constant amortized time for insertion. It is also important that the underlying array size be reduced if it is excessively large. While it might seem intuitive to also do this if $size < a.length/2$, that cutoff value could cause the array to resize too often. As a simple illustrative example, suppose that there is a dynamic array instance with a capacity of 8 that is currently at capacity. When an element is added to the collection, the capacity would be doubled to 16. Now suppose that two elements are removed. At this point there would be a capacity of 16 with only 7 elements. If we then resized the array back to 8, then after inserting 2 more elements it would need to grow, and so on. To avoid this behavior, the underlying array size is reduced by half if $size < a.length/4$. While this causes a worst-case space usage of $4n$, such a situation would be uncommon. Moreover, by not reducing the size of the array until $size < a.length/4$, we guarantee that a resizing operation that must copy x elements will only occur after at least x calls to add or remove elements. As an introduction to amortized analysis, Section B.5 proves that the resizing methods have constant amortized time complexity.

While it is important that the dynamic array automatically reduces the capacity of the underlying array when it is significantly larger than needed, it is equally important that there be a mechanism for the application program to make the underlying array significantly oversized if it is known that a large number of insertions are about to be made. This allows the application program to avoid the cost of repeated dynamic resizing. We have chosen the following semantics. The dynamic array will maintain a minimum capacity no less than that specified in the most recent call to *ensureCapacity* (or the initial capacity if there has been no call to *ensureCapacity*). However, each call to *trimToSize* not only resizes the array to the exact size of the collection, but it also has the effect of calling *ensureCapacity* with a parameter of 0.

Instance Variables and Constants: Three new instance variables are added: *minCapacity* holds minimum capacity as described above, *lowWaterMark* gives the value for *size* when the array is at its minimum allowed utilization, and *highWaterMark* gives the value for *size* when the array is at its maximum allowed utilization.

```
int minCapacity;           //capacity from last ensureCapacity (or initial capacity)
int lowWaterMark;          //size < lowWaterMark triggers shrinking
int highWaterMark;         //size > highWaterMark trigger growing
```

Abstraction Function: The abstraction function for the dynamic array is the same as that for the array. That is, for `DynamicArray` instance a ,

$$AF(a) = \langle u_0, u_1, \dots, u_{size-1} \rangle \text{ such that } u_p = a[p].$$

13.3 Representation Properties

The representation properties maintained by `Array` are all inherited by `DynamicArray`. In addition, the following properties relate to automatic resizing. These properties provide a guarantee that the underlying array length is never greater than four times the size of the collection unless the user has specified a larger capacity via *ensureCapacity*.

THRESHOLD The instance variable *lowWaterMark* maintains the minimum number of elements that should be held in a , and *highWaterMark* maintains the maximum number of elements that can be held in a . That is, $lowWaterMark = \lceil a.length/4 \rceil$ and $highWaterMark = a.length$.

PROPORTIONALITY The size of the collection is between *lowWaterMark* and *highWaterMark* (inclusive), unless an application program call to *ensureCapacity* has forced the number of elements held to go below the *lowWaterMark*. That is, $size \leq highWaterMark$, and either $size \geq lowWaterMark$ or $array.length = minCapacity$.

MINCAPACITY The capacity of a is at least as that specified by the application program using *ensureCapacity*. More formally. $a.length \geq minCapacity$.

Observe that the requirement for the instance variable *minCapacity* is that it holds the most recent parameter value for *ensureCapacity* (or the initial capacity if there have been no calls to *ensureCapacity*), unless *trimToSize* has been called more recently than *ensureCapacity* in which case *minCapacity* is 0. It is easily seen that this requirement is adhered to in the constructor, *ensureCapacity* and *trimToSize*.

13.4 Methods

We now show the internal and public methods for `DynamicArray` that are not inherited from `Array`.

13.4.1 Constructors

The most general constructor takes *capacity*, the desired initial capacity for the underlying array, and *equivalenceTester*, a user-provided equivalence tester, creates a dynamic array with the given capacity that uses the provided equivalence tester. It throws an *IllegalArgumentException* when $capacity < 0$.

```
public DynamicArray(int capacity, Comparator<? super E> equivalenceTester) {
    super(capacity);
    minCapacity = capacity;
}
```

Correctness Highlights: By the correctness of *resizeArray*, which is overridden here and called by the super constructor, THRESHOLD is satisfied. Also, the instance variable *minCapacity* is set to specification. PROPORTIONALITY and MINCAPACITY are satisfied since by the correctness of the superclass constructor, *a.length = capacity*. The rest of the correctness follows from the correctness of the Array constructor. ■

Several additional convenience constructors are provided to replace some parameters by the default values. Their correctness follows from that of the above constructor. The constructor with no arguments creates an underlying array *a* with a default initial capacity.

```
public DynamicArray(){
    this(DEFAULT_CAPACITY);
}
```

The constructor that takes a single argument, *capacity*, the initial capacity for the underlying array *a*, and creates an underlying array *a* with the given capacity. It throws an *IllegalArgumentException* when *capacity* < 0.

```
public DynamicArray(int capacity){
    this(capacity, Objects.DEFAULT_EQUIVALENCE_TESTER);
}
```

13.4.2 Representation Mutators

The only methods that must be overridden are those that could change *minCapacity*, those that resize the array (since that changes the value of the high and low water marks), and the methods that add or remove elements from the collection since they may require the array to be resized to maintain PROPORTIONALITY.

Recall that the internal method *resizeArray* takes *desiredCapacity*, the desired capacity of the underlying array, and changes the capacity of the underlying array to *desiredCapacity* while maintaining the same positional collection. It throws an *IllegalArgumentException* when executing it would violate CAPACITY.

```
protected void resizeArray(int desiredCapacity){
    super.resizeArray(desiredCapacity);
    lowWaterMark = (int) Math.ceil(a.length/4);
    highWaterMark = a.length;
}
```

Correctness Highlights: By definition, THRESHOLD is preserved. PROPORTIONALITY is preserved by the methods that call *resizeArray*. ■

Recall that *ensureCapacity* takes *capacity*, the desired capacity, and increases the capacity of the underlying array, if needed.

```

public void ensureCapacity(int capacity) {
    minCapacity = capacity;
    if (a.length < capacity)
        super.ensureCapacity(capacity);
}

```

Correctness Highlights: Also, PROPORTIONALITY is preserved since when the array is resized (by *ensureCapacity*), $a.length = minCapacity$. MINCAPACITY is preserved since *minCapacity* is set to the given capacity. Finally, THRESHOLD is maintained by the *resizeArray* method. The rest of the correctness follows from the Array *ensureCapacity* method. ■

13.4.3 Content Mutators

The only other methods that must be modified are those that add or remove elements from the collection.

Methods to Perform Insertion

All of the methods that insert elements do so by making an appropriate call to the *addImpl* method that takes p , a valid user position, and *value*, the object to insert. It inserts object *value* at position p and increments the position number for the elements that were at positions $p, \dots, size - 1$. It returns *null* for an untracked implementations. For tracked implementations, it will be overridden to return a tracker. The only change required from the superclass method is to first check if the underlying array has reached the *highWaterMark* (i.e., it is full). If so, then the capacity of the underlying array is doubled.

```

protected PositionalCollectionLocator<E> addImpl(int p, Object value) {
    if (size == highWaterMark)
        resizeArray(2*size);
    return super.addImpl(p, value);
}

```

Correctness Highlights: We consider when $size = highWaterMark$. THRESHOLD is maintained by *resizeArray*. We now argue that PROPORTIONALITY is maintained. After *resizeArray* completes $a.length = 2size = 2(size' - 1) \leq 4size'$. Thus it follows that $\lceil a.length/4 \rceil \leq size$ as required. MINCAPACITY is clearly satisfied since the capacity of the array is increased.

The rest of the correctness follows from the correctness of the superclass *addImpl* method. ■

Methods to Perform Deletion

Since all Array methods that remove elements do so by making an appropriate call to *removeRange*, we can just override this method. Recall that *removeRange*, takes as input *fromPos*, a valid position, and *toPos*, a valid position. This method requires $0 \leq fromPos \leq toPos < size$. It removes the elements at positions *fromPos*, $\dots$, *toPos*, inclusive, from the collection and decrements the positions of the elements at positions *toPos*+1 to *size*-1 by *toPos*-*fromPos*+1 (the number of elements being

removed). It throws a *PositionOutOfBoundsException* when either of the arguments is not a valid position, and it throws an *IllegalArgumentException* when *fromPos* is greater than *toPos*.

For this method, we first use the superclass *removeRange* to perform the operation. Then if *size* is at the low water mark (which occurs when the array is at most $\frac{1}{4}$ utilized), the capacity of the array is reduced by half or to the value of *minCapacity* if that is larger than half the current capacity. While it might seem intuitive to reduce the underlying array size when less than $\frac{1}{2}$ is utilized, the example on page 186 illustrates why such an approach would lead to a linear amortized cost for resizing.

```
public void removeRange(int fromPos, int toPos) {
    super.removeRange(fromPos, toPos);
    if (size ≤ lowWaterMark)
        resizeArray(Math.max(2*size, minCapacity));
}
```

Correctness Highlights: THRESHOLD is preserved by *resizeArray*. It is easily seen that MINCAPACITY is preserved since *resizeArray*, if called, guarantees that the parameter given is at least *minCapacity* and hence upon completion it also holds that $a.length \geq minCapacity$. We now argue that PROPORTIONALITY is preserved. If the array is not resized then this trivially holds, so we consider when the array is resized. If $2 \cdot size' \geq minCapacity$ then $a'.length = 2 \cdot size' \leq 4 \cdot size'$. Otherwise, $a'.length = minCapacity$ and hence PROPORTIONALITY is maintained. The rest of the correctness follows from the correctness of the superclass *removeRange* method. ■

13.5 Performance Analysis

We now argue that for any sequence of dynamic array method calls in which resizing is only performed from within the *DynamicArray add(p,value)* or *removeRange* methods, the resizing methods have constant amortized complexity. Note that any user method to add elements to the collection will call the *DynamicArray add(p,value)* method and any user method to remove elements from the collection will call the *DynamicArray remove* method. What is excluded here is any sequence of dynamic array method calls that include a call to *ensureCapacity* or *trimToSize* that have caused resizing to occur explicitly. The advantage of using a dynamic array is that the user is not responsible for these calls, so the expectation is that the user would only use these sparingly when it is known that they are not going to cause unnecessary resizing.

For the remainder of this section, we consider an arbitrary sequence of method calls that satisfies the condition given above. We argue that each call to *resizeArray* has constant amortized cost, and that there is only a constant increase in the amortized cost for the methods that add or remove elements from the collection. We use the **potential method** to compute the amortized cost. Appendix B reviews amortized complexity and methods of doing the analysis including the potential method. As part of that chapter we do a very similar analysis to that presented here where the only operations are to add an element to the end of a dynamic array, or to remove an element from the end of a dynamic array. We define a potential function Φ , which is implicitly a function of a dynamic array a , as

$$\Phi = \begin{cases} 2n - a.length & \text{if } n \geq a.length/2 \\ a.length/2 - n & \text{if } n \leq a.length/2 \end{cases}$$

where n is the number of elements held in the array. Intuitively, Φ keeps track of the “credits” available from insertions and removals that can be saved to “pay” for occasional resizing operations. When the array is at least half full, we have two credits for each element in the second half, which can be used to pay for growing the array. When the array is less than half full, we have one credit for each empty slot in the first half, which can be used to pay for shrinking the array. Observe that $\Phi = 0$ when $n = a.length/2$.

As described in Appendix B, the amortized cost of an operation is the actual cost plus the change in the potential. For the purposes of this analysis, we separate the cost of *resizeArray* from the add or remove method that called it. We first prove that the amortized cost of each call to *resizeArray* is 0, and then argue that increase in potential for any method that adds an element to a dynamic array is at most 2, and the increase in potential for any method that removes an element from the dynamic array is at most 1. Together these statements imply that the amortized cost for all public method to add or remove elements from a dynamic array (including the resizing performed) is asymptotically equal to the actual cost since it will just be the actual cost (excluding resizing) plus a small additive constant.

The table below summarizes the analysis to compute the change in potential when adding an element, removing x elements for $1 \leq x \leq n$, for doubling the array capacity (as performed by the dynamic array add methods), and for halving the array capacity (as performed by the dynamic array remove methods).

method	conditions	old Φ	new Φ	$\Delta\Phi$
add (one element)	$n \geq a.length/2$	$2n - a.length$	$2(n+1) - a.length$	+2
	$n < a.length/2$	$a.length/2 - n$	$a.length/2 - (n+1)$	-1
remove (x elements)	$n > a.length/2$	$2n - a.length$	$2(n-x) - a.length$	-2x
	$n \leq a.length/2$	$a.length/2 - n$	$a.length/2 - (n-x)$	-x
double capacity	$n = a.length$	$2n - a.length = n$	$2n - 2n = 0$	-n
half capacity	$n = a.length/4$	$a.length/2 - n = n$	$a.length/2 - n = 0$	-n

So that amortized cost for any of the public methods to add an element is at most the actual cost +2, and the amortized cost for any public methods to remove an element is at most the actual cost +1. Since the actual cost to double the array capacity is $O(n)$ and the potential is reduced by n , the amortized cost is 0. Likewise, the actual cost to half the array capacity is $O(n)$ and the potential is reduced by n , giving an amortized cost of 0.

Thus if one considers the amortized time complexities for all methods that insert or remove elements, then the asymptotic time complexities of all positional collection methods are as given in Table 11.8 and locator methods are as given in Table 11.9.

13.6 Dynamic Circular Array

package collection.positional

AbstractCollection<E> implements Collection<E>

└─ CircularArray<E> implements PositionalCollection<E>

└─ DynamicCircularArray<E> implements PositionalCollection<E>

Uses: Java primitive array

Used By: Buffer (Chapter 17)

Strengths: Random access in constant time by position is supported and also the array is automatically resized in such a way to guarantee constant amortized cost for the portion of insert or remove associated with resizing. Thus in amortized constant time an element can be added or removed from the front or back of the collection.

Weaknesses: There is a small amount of overhead required for the automatic resizing and some overhead associated with using a circular array (which enables the constant time add or remove at the front of the collection).

Critical Mutators: *add* (except when $p = \text{size}$), *addFirst*, *addAll*, *clear*, *heapsort*, *insertionsort*, *mergesort*, *quicksort*, *radixsort*, *removeFirst*, *removeRange* (except when $\text{toPosition} = \text{size} - 1$), *repositionElementByRank*, *retainAll*, *swap*, *treesort*

Competing Data Structures: If the number of objects that will be held in the collection is known when the collection is defined, then resizing is not needed and using a circular array will reduce overhead slightly. Also, if objects are not inserted or removed in the front portion of the collection, then a non-dynamic circular array can be used. Finally, if a tracker versus a marker is needed then the tracked array, singly linked list, or doubly linked list should be used.

Internal Representation: *DynamicCircularArray* extends *CircularArray* with the only change being a modification to methods that add or remove items to first check if resizing is needed. This is done in the same manner as with the dynamic array except that here *CircularArray* is extended.

Performance Analysis: As we argued in Section 13.5 the amortized cost for the resizing is constant. Thus if one considers the amortized time complexities for all methods that insert or remove elements, then the asymptotic time complexities of all methods are given in Table 12.4.

The implementation is analogous to *DynamicArray*, and is included on the accompanying CD.

Chapter 14

Tracked Array Data Structure

package `collection.positional`

`AbstractCollection<E>` implements *Collection<E>*

└ `Array<E>` implements *PositionalCollection<E>*

└ `CircularArray<E>` implements *PositionalCollection<E>*

└ `DynamicCircularArray<E>` implements *PositionalCollection<E>*

└ **`TrackedArray<E>`** implements *PositionalCollection<E>*, *Tracked<E>*

Uses: Java primitive array

Used By: `BinaryHeap` (Chapter 25)

Strengths: This data structure provides a tracked implementation of a dynamic circular array. As discussed in Section 5.8, tracked versions of data structures are useful when the application needs to locate items in the collection in constant time, even if they might have been moved. Also, trackers support iteration that is robust to most concurrent modifications. Furthermore, even when a critical mutation is executed, such as sorting the collection, each tracker continues to track the same element as before the mutation. In this chapter, we provide an example of extending `DynamicCircularArray` to provide a tracked version of a dynamic circular array. However, substituting a different parent class name (`Array`, `CircularArray`, or `DynamicArray`) yields a tracked version of the corresponding data structure.

Weaknesses: As for `DynamicCircularArray`, it takes linear time to add or remove an element for the middle portion of the array, even if a locator is used to locate the element. There is both time and space overhead to support the tracker. In particular, whenever an element is swapped or moved, its locator must be updated to reflect the new position. To accomplish this, each array slot contains an object (which we call a **node**) with both a reference to the element and its current index within the array. The tracker encapsulates a reference to that node. Even if the user does not retain a reference to the tracker, the implementation still incurs the overhead of maintaining the node. However, the overhead is small and does not change the asymptotic time complexity for any method. The space overhead is the allocation of one node instance per element.

Critical Mutators: *heapsort*, *insertionsort*, *mergesort*, *quicksort*, *radixsort*, *repositionElementByRank*, *swap*, *treesort*

Competing Data Structures: If all elements are added or removed from one end of the collection, it would be more efficient to use a version of this class that extends `DynamicArray` instead of `DynamicCircularArray`. (This can be accomplished simply by changing the *extends* clause in the class header.) If constant time access via position is not required and elements are to be added or removed at the middle portion of the collection (via a locator), then a list-based approach is a better option. Finally, if the application does not need to track a particular element, and does

not require continued iteration after an element is added or removed, the tracking overhead of this implementation would be wasteful, and an untracked implementation would be a more efficient choice.

14.1 Internal Representation

The primary change in a tracked array is a level of indirection within the underlying array to support the maintenance of trackers. Instead of holding a direct reference to an element, each slot in the underlying array holds a reference to a *Node* instance (defined below) which, in turn, has a reference to the element. Besides holding the element reference, the primary purpose of the node is to keep track of that element's index within the array. Whenever an element is moved, the index stored within the node is updated. The trackers for an element encapsulate a reference to that element's node. Since the index in the node is kept up to date, the tracker can always determine, in constant time, where the element is located within the array. (Note that because all trackers for an element refer to the same node, only one update is required for a moved element, regardless of the number of trackers.)

The other purpose of the node concerns the removal of an element from the collection. To prevent a tracker from accessing a removed element, one could imagine storing a boolean value in the node to indicate whether or not its element is still in the collection. However, recall that the semantics of a tracker is that when its element is removed from position p , the tracker is considered to be *between* the elements that were at positions $p - 1$ and $p + 1$. Moreover, note that after an element is removed from the collection, its node's index will no longer be meaningful and will not be updated as elements are rearranged within the array. Nonetheless, for the tracker t of a removed item to continue to be used as an iterator, it must be able to determine the index of the element that was at position $p - 1$ when its element was removed. Furthermore, if that element is also removed, there must be a mechanism for finding the index for its predecessor, and so on. To support this, each node contains a *redirect* reference, which is *null* as long as the element is still in the collection. If the element tracked by t has been removed from the collection then *redirect* references the node that precedes that node in the collection. If the element for that predecessor node is subsequently deleted then its *redirect* reference can be followed. This process can be repeated until reaching a node with a *null* value for *redirect*, which is guaranteed to be the predecessor of t in the iteration order.

We say that the sequence of *redirect* references beginning at a node is its **redirect chain**, and that the **end of the redirect chain** is the node with the *null redirect* reference that is reached by following all of the references in that chain. Note that the dummy node for FORE always has a null redirect reference, so if there are no predecessors for a tracker, its redirect chain will end at FORE. As with the **path compression** used for the union-find data structure (Section 6.3), to optimize performance, redirect chains are compressed whenever they are traversed. Also note that if the application retains no trackers that refer into a given chain, all nodes in that chain become garbage, so no space overhead is incurred for the removed items.

Instance Variables and Constants: For the array-based implementations we have presented so far, we used the constants *FORE* for position held within a marker at FORE, and *size* for the position held within a marker at AFT. Since a tracker will hold a reference to a node, versus a position, we replace the roles of FORE and AFT by the node references *FORE_NODE* and *AFT_NODE*.

```
static final Node FORE_NODE = new Node(-1, null);
static final Node AFT_NODE = new Node(-1, null);
```

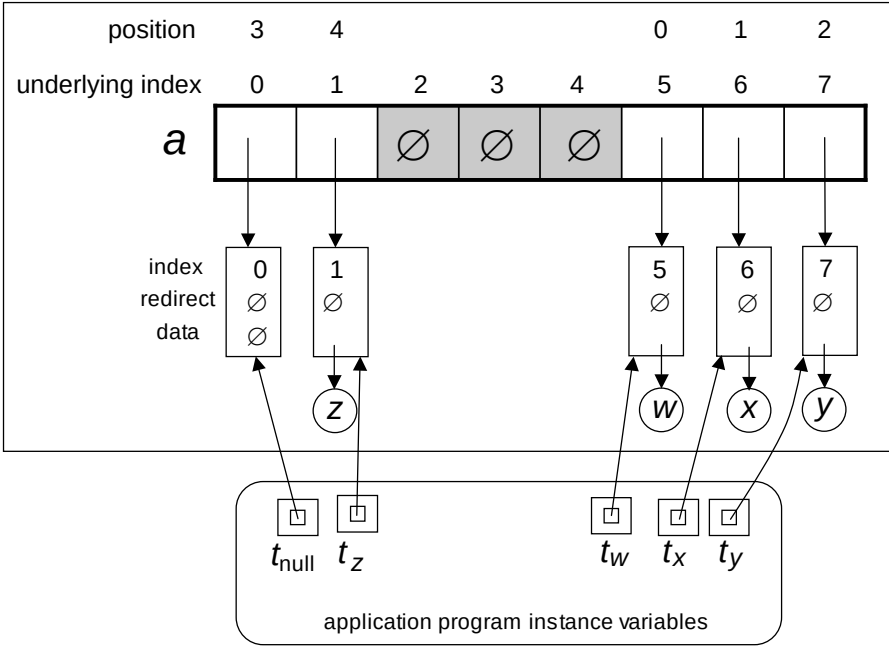


Figure 14.1
A populated example of a tracked array for the positional collection $\langle w, x, y, null, z \rangle$. A tracker is shown for all five elements in the collection.

No other instance variables are added. However, each slot of the underlying array references a node instead of an element.

Populated Example: Figure 14.1 shows the internal representation for a tracked array for the positional collection $\langle w, x, y, null, z \rangle$. In this example, the application has retained a tracker for each element of the collection, providing it with the ability to locate any element in constant time, even if elements have been rearranged (e.g., by sorting the collection). Although the application in this example has retained trackers for all elements, some applications may retain trackers for only certain elements, for the purpose of iteration, or possibly for navigation in the vicinity of the tracked elements.

Figure 14.2 shows the internal representation after x is removed (via $t_x.remove()$, $remove(1)$, or any other way it could be removed). In this example, t_x is between positions 0 and 1, so the removed node is between positions 0 and 1. Note that the end of the redirect chain for node x is the node at position 0. Therefore, if $t_x.advance()$ is subsequently called, that tracker would be positioned at element y .

Abstraction Function: The abstraction function for a tracked array differs from that of a dynamic circular array by taking the level of indirection into account. Let TA be a tracked array. The abstraction function is

$$AF(TA) = \langle u_0, u_1, \dots, u_{size-1} \rangle \text{ such that } u_p = a[(start + p) \% a.length].data.$$

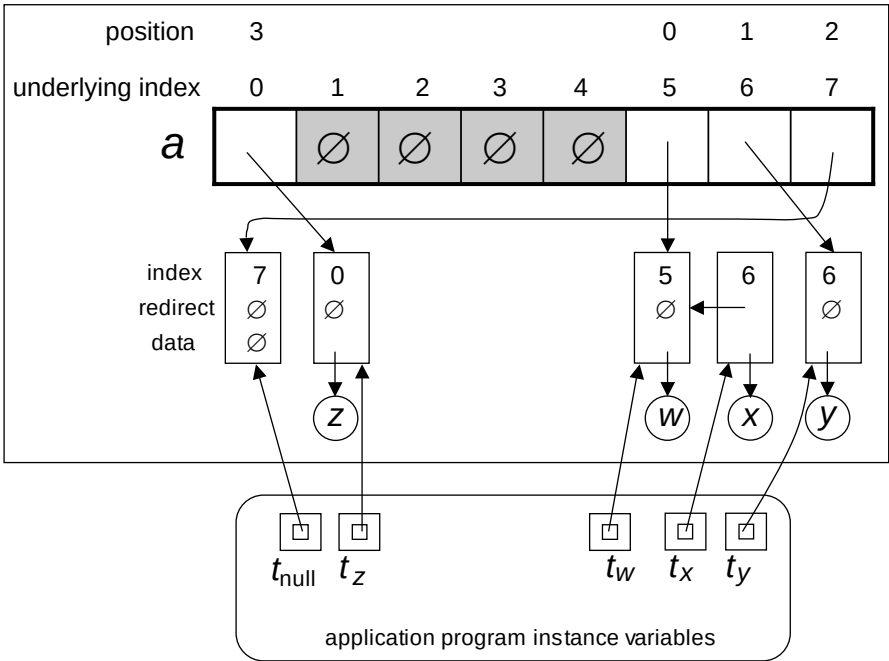


Figure 14.2
The tracked array that results when *x* is removed from the tracked array shown in [Figure 14.1](#).

Terminology: Throughout this section we need a succinct and unambiguous way to discuss the location of a tracker and the relationship between a node and the element that it holds. Recall that when a tracked element is removed, logically the tracker is now between two elements in the collection (but not at any element). For ease of exposition, we use the following definitions throughout this chapter.

- If $a[(start + p) \% a.length] = x$ then we say that node *x* is **at position** *p*.
- If node *x* is at position *p*, and then the element at position *p* is removed from the collection, we say that node *x* is **between positions** *p* − 1 and *p*.
- The position of a tracker is the same as the position of the node to which it refers. That is, if tracker *t* refers to a node at position *p*, then we say that tracker *t* is **at position** *p*. Similarly, if tracker *t* refers to a node between positions *p* − 1 and *p*, we say that tracker *t* is **between positions** *p* − 1 and *p*.

To illustrate these definitions, we note a few examples from the figures. Tracker t_z in Figure 14.1 is at position 4 because the node it refers to is in slot 1 of the array, which corresponds to user position 4. As another example, tracker t_x in Figure 14.2 is between positions 0 and 1 since the redirect chain for its node ends with the node at position 0. In both examples shown in [Figure 14.3](#), t_w is at position 0, t_z is at position 1, and t_x , t_y , and t_n are between positions 0 and 1. Thus, if the application repeatedly removes the current element at position *p*, several trackers may be between the same two positions.

Note that the index retained in the node is the underlying array index, not the user position. To illustrate why storing the user position would be a bad idea, consider when a new element is added to position 0. Since the underlying array is circular, this can be achieved by just inserting the item

in the underlying array slot just before the element currently at position 0, and updating the instance variable recording the index where the array starts. This method takes constant time. While the index for the existing elements has not changed, the position for each increases by one. If the position were stored in each node then all existing nodes would need to be updated, which would increase the time complexity to linear time. By storing the underlying index, this method can still run in constant time.

Design Notes: This class illustrates how most of the changes required to support trackers can be supported by creating an object that both references the element and also holds its index in the underlying array. This allows any methods that move an element to instead move the node, and also to update the index so that a tracker to the node can access the appropriate array position in constant time. The only time cost of this indirection is that one additional reference must be followed to retrieve the element itself.

While each sorting algorithm must invalidate all trackers for iteration, our implementation ensures that each tracker continues to track the same element even after the collection has been sorted. We achieve this task, by using the **wrapper design pattern** to wrap the user-provided comparator within a comparator that is defined over nodes, and then the sorting algorithms from Array can be applied directly to sort the nodes. Likewise, the user-provided bucketizer or digitizer can be wrapped to be defined over the nodes.

Optimizations: Since path compression is applied to the redirects chain as they are traversed, each redirect reference is followed at most once. Thus, the time complexity overhead for redirect references is negligible. However, they do increase the space usage. If it is known that an application will never remove an item during iteration, then the *redirect* field and the code that uses it could be eliminated. If such a change were made, *remove* would become a critical mutator, and should call *version.increment()* to invalidate the active trackers for iteration. Finally, if this change were made, the Tracker *remove* method would need to restore its own version number so that removes done by a tracker would not result in a concurrent modification exception within that tracker.

14.2 Representation Properties

We inherit the CAPACITY and NONRETENTION properties from Array, and the THRESHOLD, PROPORTIONALITY, and MINCAPACITY properties from DynamicArray. The changes here do not affect the maintenance of any of these properties, so we do not discuss them further.

We replace PLACEMENT to address the extra level of indirection, and introduce three new properties.

PLACEMENT: Each node in the array refers to the proper element. More formally, $a[(start + p) \% a.length].element = u_p$ for all $p \in \{0, \dots, size-1\}$.

SELFREFERENCE: The node for each element refers to its own index in the underlying array. More formally, if i is in use, $a[i].index = i$.

INUSE: The redirect reference is *null* for all elements in use. That is, $a[i].redirect = \text{null}$ for all i that are in use.

REDIRECTCHAIN: Let x be a node between positions $p - 1$ and position p . The chain of *redirect* references starting from x ends at the node at position $p - 1$.

14.3 Node Inner Class

Node<*E*>

Each node includes three instance variables: *index*, the underlying index in the array, *redirect*, the head pointer for the redirect chain, and *data*, a reference to the element held in the node.

```
static class Node<E> {
    int index;           //satisfies, a[node.index] = node for underlying array a
    Node<E> redirect;    //head pointer for redirect chain, or null if in the collection
    E data;              //the element held in this node
}
```

The constructor takes *index*, the index in the underlying array, and *element*, the element to hold in the node. It initializes the new node with a *null* redirect reference, which indicates that *element* is in the collection.

```
Node(int index, E element) {
    this.index = index;
    data = element;
    redirect = null;
}
```

An obvious effect of using these nodes in the implementation is a level of indirection for element access.

14.4 Tracked Array Methods

We show the internal and public methods for the tracked array data structure that are not inherited from *DynamicCircularArray* (or whichever other array data structure is chosen for the tracked array to extend).

14.4.1 Constructors

The most general constructor takes *capacity*, the desired initial capacity for the underlying array, and *equivalenceTester*, a user-provided equivalence tester, creates a tracked array with the given capacity that uses the provided equivalence tester. It throws an *IllegalArgumentException* when *capacity* < 0.

```
public TrackedArray(int capacity, Comparator<? super E> equivalenceTester) {
    super(capacity, equivalenceTester);
}
```

Correctness Highlights: Follows immediately from that of the *DynamicCircularArray* constructor and the fact that *PLACEMENT*, *SELFREFERENCE*, *INUSE*, and *REDIRECTCHAIN* hold since no indices are in use and there are no nodes referenced by a tracker (since there are no nodes prior to the first insertion). ■

Two additional convenience constructors are provided to replace some parameters by the default values. Their correctness follows from that of the above constructor. The constructor with no arguments creates an underlying array *a* with a default initial capacity.

```
public TrackedArray(){
    this(DEFAULT_CAPACITY);
}
```

The constructor with a single argument, *capacity*, the initial capacity for the underlying array *a*, throws an *IllegalArgumentException* when *capacity* < 0.

```
public TrackedArray(int capacity){
    super(capacity);
}
```

14.4.2 Representation Accessors

Along with the representation accessors inherited from *DynamicCircularArray*, we introduce methods to access the elements held within the nodes. We have not overridden *read* or *write* since methods are needed that directly access the contents of the array slots. As discussed in the design notes in Chapter 11, for the sake of efficiency no checks are performed within internal methods that take a valid position. It is expected that the public method perform such checks, and throw the appropriate exception.

The method *readElement* takes *p*, a valid position, and returns the element held at user position *p*. It requires that *p* is a valid position.

```
protected E readElement(int p) {
    return ((Node<E>) read(p)).data;
}
```

| *Correctness Highlights:* The correctness follows from that of *read* and *PLACEMENT*. ■

The method *writeElement* takes *p*, a valid position, and *element*, the element to write at position *p*, and returns the element that was previously at position *p*. This method requires that *p* is a valid position.

```
protected E writeElement(int p, E element) {
    Node<E> t = (Node<E>) read(p);
    E oldElement = t.data;
    t.data = element;
    return oldElement;
}
```

| *Correctness Highlights:* The correctness follows from that of *read* and *PLACEMENT*. ■

14.4.3 Algorithmic Accessors

Two algorithmic accessors need to be modified to compensate for the extra level of indirection. Recall that the public *get* method takes *p*, a valid user position, and returns the element at the position *p*. It throws a *PositionOutOfBoundsException* when the position is not valid.

```

public E get(int p) {
    if (p < 0 || p > size - 1)
        throw new PositionOutOfBoundsException(p);
    return readElement(p);
}

```

Correctness Highlights: The correctness follows from PLACEMENT, and the correctness of *readElement*. ■

Recall, *findPosition* takes *element*, the target, and returns the first position *p* in the collection holding an equivalent element, or *NOT_FOUND* if there is no equivalent element in the collection.

```

protected int findPosition(E element) {
    for (int p = 0; p < size; p++)
        if (equivalent(element, readElement(p)))
            return p;
    return NOT_FOUND;
}

```

Correctness Highlights: Follows from PLACEMENT, SIZE, and the correctness of *readElement*. After all positions are considered, if the element has not been found then it is not in the collection. ■

14.4.4 Representation Mutators

We first describe an internal method to update the index within the node to maintain SELFREFERENCE when nodes are shifted the array. The *updateNodes* method takes *i*, the starting index that needs updating, and *num*, the number of nodes to update starting at *i*, possibly wrapping.

```

protected void updateNodes(int i, int num) {
    for (int m = 0; m < num; m++){
        ((Node) a[i]).index = i;
        i = nextIndex(i);
    }
}

```

Correctness Highlights: By the correctness of *nextIndex* and PLACEMENT, this method ensures SELFREFERENCE is satisfied for *num* consecutive indices starting at the specified index. ■

Recall that *shiftLeft* takes *p1*, the starting position to shift, *p2*, the ending position for the shift, and *num*, the number of slots to shift. It shifts the elements held in *p1*, ..., *p2* (possibly wrapped) left *num* slots. It will overwrite the *num* positions left of *p1*. It requires that $p1 \leq p2$, and $0 < num \leq a.length - (p2 - p1 + 1)$ (the amount of the shift is at least 1, and is not so much that the portion of the array being shifted would intersect with itself).

```

protected void shiftLeft(int p1, int p2, int num) {
    super.shiftLeft(p1, p2, num);
    //perform the shift
}

```

```

    int i = getIndex(p1) - num;
    if (i < 0)
        i += a.length;
    updateNodes(i, p2-p1+1);
}

```

Correctness Highlights: Observe that the starting new index for the shifted portion of the collection is the current index for $p1$ minus the number of slots shifted. Since the array is circular, this index may wrap from the front to the back. Mathematically, this corresponds to adding $a.length$ if the new index is negative. By the correctness of *updateNodes*, SELFREFERENCE is preserved. The rest of the correctness follows by that of the inherited *shiftLeft* method. ■

Recall that *shiftRight* takes $p1$, a valid position, $p2$, a valid position, and num , the number of positions to shift. It shifts the elements held in $p1$, ..., $p2$ (possibly wrapped) right num slots. It will overwrite the num positions right of $p2$. This method requires that $p1 \leq p2$, and $0 < num \leq a.length - (p2 - p1 + 1)$ (the amount to shift is at least 1, and the portion of the array being shifted does intersect with itself). This method works in a symmetric manner to *shiftLeft*.

```

protected void shiftRight(int p1, int p2, int num) {
    super.shiftRight(p1, p2, num);           //perform the shift
    updateNodes(getIndex(p1+num), p2-p1+1);
}

```

Correctness Highlights: It is analogous to *shiftLeft* with the only differencing being that the new starting index for the shifted portion of the collection will be at the current index for $p1$ plus the number of slots shifted. Since the array is circular, this index may wrap from the back to the front. However, since *getIndex* already handles this case, it can be used to compute the starting index for the shifted portion. ■

Recall that the internal method *resizeArray* takes *desiredCapacity*, the size to make the underlying array, and changes the size of the underlying array to *desiredCapacity* while maintaining the same positional collection. It throws an *IllegalArgumentException* when executing it would make the array capacity too small to hold the current collection.

```

protected void resizeArray(int desiredCapacity) {
    super.resizeArray(desiredCapacity);
    updateNodes(0, size);
}

```

Correctness Highlights: By the correctness of *updateNodes*, SELFREFERENCE is preserved. The rest of the correctness follows from that of *resizeArray*. ■

14.4.5 Content Mutators

The simplest content mutator, *set*, takes p , a valid user position to be updated, and *element*, the element to put at position p , and returns the prior element at position p . It throws a *PositionOutOfBoundsException* when p is not a valid position. As with the algorithmic accessors, the only change required is to replace the use of *write* with *writeElement*.


```

public E set(int p, E element) {
    if (p < 0 || p > size - 1)
        throw new PositionOutOfBoundsException(p);
    return writeElement(p, element);
}

```

Recall that the internal *swapImpl* method takes *pos1*, a valid position, and *pos2*, a valid position. It swaps the elements held in positions *pos1* and *pos2*. Note that the inherited *swapImpl* method invalidates all active locators for iteration. While all trackers continue to track the same elements, if used for iteration a persistent element could be seen twice, or not at all. Thus, this method remains a critical mutator.

```

void swapImpl(int pos1, int pos2) {
    super.swapImpl(pos1, pos2);           //swaps and invalidates locators for iteration
    int index1 = getIndex(pos1);
    int index2 = getIndex(pos2);
    ((Node) a[index1]).index = index1;   //preserve SelfReference
    ((Node) a[index2]).index = index2;   //preserve SelfReference
}

```

Correctness Highlights: The last two lines directly preserve SELFREFERENCE. The rest of the correctness follows from that of *getIndex* and the superclass *swapImpl* methods. ■

The *move* method takes *fromPos*, the current position of the element to move, and *toPos*, the position where the element is to be moved. It requires that *fromPos* and *toPos* are both legal positions.

```

void move(int fromPos, int toPos) {
    int toIndex = getIndex(toPos);
    a[toIndex] = a[getIndex(fromPos)];           //move the node
    ((Node) a[toIndex]).index = toIndex;         //preserve SelfReference
    version.increment();                         //invalidate all trackers for iteration
}

```

Correctness Highlights: The second to last line preserves SELFREFERENCE for the node that is moved. Since this mutation may cause the element being moved to be seen twice or not at all during iteration, all active trackers must be invalidated. The rest follows from the correctness of *getIndex*. ■

The *put* method takes *p*, a valid position, and *x*, the node to place at position *p*. It requires that *p* is a legal position.

```

void put(int p, Object x) {
    int i = getIndex(p);
    a[i] = x;                                   //put x at given position
    ((Node) a[i]).index = i;                   //preserve SelfReference
}

```

Correctness Highlights: The last line preserves SELFREFERENCE. Since the elements in both x and the node currently in position p are not persistent elements, iteration through any existing markers can properly continue. The rest follows from that of *getIndex*. ■

Methods to Perform Insertion

Recall that *addImpl* takes p , a valid position, and *element*, the element to insert. It inserts *element* at position p , incrementing the position for the elements that were at positions $p, \dots, \text{size}-1$, and returns a tracker for the new element.

```
public PositionalCollectionLocator<E> addImpl(int p, Object element) {
    Node newNode = new Node(-1, element);           //temporarily put -1 for the index
    int count = version.getCount();
    super.addImpl(p, newNode);
    newNode.index = getIndex(p);                     //preserve SelfReference
    version.restoreCount(count);                     //don't need to invalidate locators
    return new Tracker(newNode);
}
```

Correctness Highlights: PLACEMENT is preserved based on the correctness of the superclass *add* method and the *Node* constructor. By the correctness of *add* (which places the element at position p), and the *getIndex* method, SELFREFERENCE holds. Since the *Node* constructor initializes the *redirect* field to be *null*, INUSE holds. Finally, REDIRECTCHAIN trivially holds, since the new node is in use. By restoring the modification count to its value before the superclass *addImpl* method was executed, all locators remain valid. ■

The *addTracked* method from the *Tracked* interface takes *value*, the new element, and inserts *value* it at the end of the collection.

```
public PositionalCollectionLocator<E> addTracked(E value) {
    return addImpl(size, value);                     //adds to end
}
```

Correctness Highlights: Follows from that of *addImpl*. ■

Finally, we also include a public *addTracked* method that takes p , a valid user position, and *value*, the new element. It inserts the new element at position p and increments the position number for the elements that were at positions $p, \dots, \text{size} - 1$. It throws a *PositionOutOfBoundsException* when p is neither *size* nor a valid position.

```
public PositionalCollectionLocator<E> addTracked(int p, E value) {
    return addImpl(p, value);
}
```

Correctness Highlights: Follows from that of *addImpl*, which throws an exception if p is not *size* or a valid position. ■

Methods to Perform Deletion

Since all methods to remove elements make an appropriate call to *removeRange*, we need to extend this method to properly set *redirect*. Recall *removeRange* takes *fromPos*, the first position to be removed and *toPos*, the last position to be removed. It removes *a[i]...a[j]* from the collection moving the elements at positions *j+1* to *size-1* to the left by *j-1+1* positions. It throws a *PositionOutOfBoundsException* when either of the arguments is not a valid position, and it throws an *IllegalArgumentException* when *fromPos* is greater than *toPos*.

```

public void removeRange(int fromPos, int toPos) {
    if (fromPos < 0 || toPos ≥ size)
        throw new PositionOutOfBoundsException();
    if (fromPos > toPos)
        throw new IllegalArgumentException();
    Node predecessor = FORE_NODE;           //predecessor for all removed elements
    if (fromPos > 0)
        predecessor = (Node) a[getIndex(fromPos - 1)];
    for (int p = fromPos; p ≤ toPos; p++)    //preserve RedirectChain
        ((Node) a[getIndex(p)]).redirect = predecessor;
    int count = version.getCount();
    super.removeRange(fromPos, toPos);
    version.restoreCount(count);           //do not need to invalidate locators
}

```

Correctness Highlights: By SIZE, the exceptions are thrown as specified. SELFREFERENCE is maintained by the *shiftLeft* or *shiftRight* method used by the superclass *removeRange* method. The rest follows from the correctness of the superclass *removeRange*.

Finally, we argue that the for loop preserves INUSE and REDIRECTCHAIN. Any trackers for elements being removed will be between position *fromPos-1* and position *fromPos* unless *fromPos* is 0, in which case the tracker will be between *FORE_NODE* and position 0. Thus to satisfy REDIRECTCHAIN all of the nodes for the removed elements must have the redirect pointer reference the node at position *fromPos-1* (or *FORE_NODE* if *fromPos* is 0). This is exactly the computation performed by the loop. By restoring the modification count to its value before the superclass *removeRange* method was executed, all locators remain valid. ■

The *clear* method removes all elements from the collection.

```

public void clear() {
    if (!isEmpty())
        removeRange(0, getSize()-1);
}

```

14.4.6 Locator Initializers

The *iterator* method returns a new tracker that is initialized to be logically just before the first element in the collection.

```

public PositionalCollectionLocator<E> iterator() {
    return new Tracker(FORE_NODE);
}

```

Correctness Highlights: Follows from the fact that the constant *FORE_NODE* is logically at the position that precedes the first element in the collection. ■

The *iteratorAtEnd* method returns a new tracker that is initialized to be logically just after the last element in the collection.

```
public PositionalCollectionLocator<E> iteratorAtEnd() {
    return new Tracker(AFT_NODE);
}
```

Correctness Highlights: Follows from the fact that the constant *AFT_NODE* is logically at the position that follows the last element in the collection. ■

The *iteratorAt* method takes *pos*, the user position of an element. It returns a new tracker that is at the given position. It throws a *NoSuchElementException* when the given position is not a valid user position.

```
public PositionalCollectionLocator<E> iteratorAt(int pos) {
    if (pos < 0 || pos ≥ size)
        throw new NoSuchElementException();
    return new Tracker((Node) read(pos));
}
```

Finally, the *getLocator* method takes *value*, the target, and returns a tracker initialized to the position of the first element in the collection equivalent to *value*. It throws a *NoSuchElementException* when *value* does not occur in the collection.

```
public PositionalCollectionLocator<E> getLocator(E value) {
    int position = findPosition(value);
    if (position == NOT_FOUND)
        throw new NoSuchElementException();
    return new Tracker((Node) read(position));
}
```

14.5 Wrappers for Sorting Tracked Arrays

The only change required to sort a tracked array is to modify the comparator so that it compares nodes by comparing their elements. Using this approach, the trackers will be preserved during the sort. We use an anonymous class in the *getSorter* method that takes *comp*, the comparator to use for the elements, and returns a comparator defined over nodes for use by the sorting algorithms.

```
private Comparator getSorter(final Comparator<? super E> comp){
    return new Comparator<Node<E>>() {           //anonymous comparator over nodes
        public int compare(Node<E> a, Node<E> b) {
            return comp.compare(a.data, b.data);
        }
    };
}
```

Each sorting algorithm (insertion sort, mergesort, heap sort, tree sort, and quicksort) that take a comparator as an argument simply calls the corresponding implementation to sort the array of nodes by wrapping the provided comparator so it is defined over the nodes.

There is only one other method that must be overridden. Recall that *buildPriorityQueue* takes *pq*, the priority queue instance to which all elements are to be placed. In the *Array* class, this method uses the *addAll* method which is a linear time method for a binary heap. However *addAll* adds each element to *pq*, whereas for the tracked array, the nodes themselves must be added. While we could repeat the binary heap *addAll* implementation here (as long as *pq* is a binary heap), we instead just successively add each node. While this method has time complexity $O(n \log n)$, the asymptotic complexity of heap sort is unaffected.

```
protected void buildPriorityQueue(PriorityQueue<Object> pq) {
    for (int i = 0; i < getSize(); i++)
        pq.add(a[getIndex(i)]);
}

public void insertionsort(Comparator <? super E> comp) {
    insertionsortImpl(getSorter(comp));
}

public void mergesort(Comparator <? super E> comp) {
    mergesortImpl(getSorter(comp));
}

public void heapsort(Comparator <? super E> comp) {
    heapsortImpl(getSorter(comp));
}

public void treesort(Comparator <? super E> comp) {
    treesortImpl(getSorter(comp));
}

public void quicksort(Comparator <? super E> comp) {
    super.quicksort(getSorter(comp));
}
```

As with the sorting algorithms, *getElementAtRank* uses the same approach of using the wrapped comparator created by *getSorter*.

```
public E repositionElementByRank(int i, Comparator <? super E> comp) {
    return (E) super.repositionElementByRank(i, getSorter(comp));
}
```

Similar to the *getSorter* method, the *getNodeDigitizer* method that takes *digitizer*, the digitizer to use for the elements uses an anonymous class to wrap *digitizer*. It returns a digitizer defined over nodes for use by radix sort. Each of the methods in the digitizer interface that takes a node *x* as an argument, replaces it by *x.element* in a call to the corresponding method of the provided digitizer, which is defined over the elements.

```
private Digitizer getNodeDigitizer(final Digitizer<? super E> digitizer) {
    return new Digitizer<Node<E>>() {                                //anonymous node digitizer class
        public int getDigit(Node<E> x, int place) {
            return digitizer.getDigit((x.data), place);
        }
        public int getBase() {
```

```

        return digitizer.getBase();
    }
    public boolean isPrefixFree(){
        return digitizer.isPrefixFree();
    }
    public int numDigits(Node<E> x) {
        return digitizer.numDigits(x.data);
    }
};
}

```

The radix sort implementation just sends the wrapped digitizer returned by *getNodeDigitizer* to the array radix sort implementation.

```

public void radixsort(Digitizer <? super E> digitizer) {
    radixsortImpl(getNodeDigitizer(digitizer));
}

```

As with *getSorter* and *getNodeDigitizer* methods, the *getNodeBucketizer* method that takes *bucketizer*, the bucketizer to use for the elements. It uses an anonymous class to wrap *bucketizer*. It returns a bucketizer defined over nodes for use by bucket sort.

```

private Bucketizer getNodeBucketizer(final Bucketizer<? super E> bucketizer) {
    return new Bucketizer<Node<E>>() {
        public int getBucket(Node<E> x) {
            return bucketizer.getBucket(x.data);
        }
        public int getNumBuckets() {
            return bucketizer.getNumBuckets();
        }
        public int compare(Node<E> a, Node<E> b) {
            return comp.compare(a.data, b.data);
        }
    };
}

```

The bucket sort implementation also sends the wrapped digitizer returned by *getNodeBucketizer* to the array bucket sort implementation.

```

public void bucketsort(Bucketizer <? super E> bucketizer) {
    bucketsortImpl(getNodeBucketizer(bucketizer));
}

```

14.6 Tracker Inner Class

`AbstractCollection<E>.AbstractLocator<E>` implements *Locator<E>*

└ **Tracker** implements *PositionalCollectionLocator<E>*

We now describe the *Tracker* inner class. It has a single instance variable, *trackedNode*, which holds a reference to the node to track (possibly *FORE_NODE* or *AFT_NODE*).

```
protected Node<E> trackedNode;
```

We replace the `MARKERLOC` property of the `Array BasicMarker` class by the following representation property.

TRACKSELEMENT: The tracked node is not *null*. That is, *trackedNode* references *FORE_NODE*, *AFT_NODE*, or a `TrackedArray Node` instance.

The constructor takes *toTrack*, the node to track. It throws an *IllegalArgumentException* when `TRACKSELEMENT` would be violated.

```
protected Tracker(Node<E> toTrack) {
    if (toTrack == null)
        throw new IllegalArgumentException();
    this.trackedNode = toTrack;
}
```

Correctness Highlights: `TRACKSELEMENT` is trivially satisfied since it is checked by the constructor. ■

The *inCollection* method returns *true* if and only if the tracked element is currently in the collection.

```
public boolean inCollection(){
    if (trackedNode == FORE_NODE || trackedNode == AFT_NODE)
        return false;
    return trackedNode.redirect == null;
}
```

Correctness Highlights: Follows from `INUSE` and `REDIRECTCHAIN`. ■

The *get* method returns the tracked element. It throws a *NoSuchElementException* when the tracker is not at an element in the collection.

```
public E get() {
    if (!inCollection())
        throw new NoSuchElementException();
    return trackedNode.data;
}
```

Correctness Highlights: Follows immediately from *inCollection*, `PLACEMENT` and `TRACKSELEMENT`. ■

Recall *set* takes *element*, the element to store at the current tracker location. It returns the element that had been stored at the tracker location and throws a *NoSuchElementException* when the tracker is not at an element in the collection.

```
public E set(E element) {
    if (!inCollection())
```

```

        throw new NoSuchElementException();
    E oldElement = trackedNode.data;
    trackedNode.data = element;
    return oldElement;
}

```

Correctness Highlights: Follows immediately from **PLACEMENT**, **TRACKSELEMENT**, and the correctness of *inCollection*. It is easily seen that **PLACEMENT** is preserved. ■

We also provide an accessor *getCurrentPosition* that returns the position of the tracker. It throws a *NoSuchElementException* when the tracker is not at an element in the collection.

```

public int getCurrentPosition() {
    if (!inCollection())
        throw new NoSuchElementException();
    return getPosition(trackedNode.index);
}

```

Correctness Highlights: Follows from **TRACKSELEMENT**, **SELFREFERENCE**, and the inherited *getPosition* method. ■

We need methods that can follow the redirect pointers to find the element in the collection that precedes an element that had been removed. For the sake of efficiency, if we follow a sequence of redirect pointers, then once a node with a *null* redirect field is found, we update all the redirect pointers that were followed to point to this field.

The internal *skipRemovedElements* method takes *ptr*, reference to a node, and returns the first element still in the collection that is reached when following the *redirect* pointers. This method performs the path compression, like that done by the union-find data structure (Section 6.3). However, the amortized analysis here is quite straightforward since each intermediate reference in the redirect chain is traversed at most once. Since exactly one reference is added to the redirect chain per element removed, *skipRemovedElements* has constant amortized cost.

```

private Node<E> skipRemovedElements(Node<E> ptr) {
    if (ptr.redirect == null)
        return ptr;
    if (ptr.redirect.redirect != null) //for efficiency
        ptr.redirect = skipRemovedElements(ptr.redirect);
    return ptr.redirect;
}

```

Correctness Highlights: By **REDIRECTCHAIN**, the first element reached with a *null redirect* field is the correct element to return. Observe that when *ptr.redirect.redirect* is not *null*, then the *redirect* field already contain the desired answer, so no recursion is needed.

This method is guaranteed to terminate since *FORE_NODE* always has a redirect field that is null. If *FORE_NODE* is the first such element found, then there are no elements that remain in the collection before the initial value sent for *ptr*. Figure 14.3 shows a tracked array before and after calling *skipRemovedElements*. The physical position of the nodes in the figure have been moved to help make it more readable. The only invariant affected is **REDIRECTCHAIN** since this method only changes the state for the elements no longer in the collection. Since all the updated

redirect pointers will now reference the end of their prior redirect chains, REDIRECTCHAIN is preserved. ■

The internal *nextCollectionElement* method returns a reference to the next node in the collection. More specifically, if *trackedNode* is either at position $p-1$ or if *trackedNode* is between positions $p-1$ and p , then a reference to the node at position p is returned. If *trackedNode* is *FORE_NODE* then the node at position 0 is returned (unless the collection is empty in which case *AFT_NODE* is returned). This method requires that *trackedNode* is not at *AFT_NODE*. For the sake of efficiency, *skipRemovedElements* is used when *trackedNode* is no longer in the collection.

```
private Node<E> nextCollectionElement() {
    if (trackedNode.redirect != null)                //tracked node not in collection
        trackedNode = skipRemovedElements(trackedNode);
    if ((trackedNode == FORE_NODE && size==0) ||
        trackedNode.index == getIndex(size-1))
        return AFT_NODE;
    if (trackedNode == FORE_NODE)
        return (Node<E>) a[getIndex(0)];
    else
        return (Node<E>) a[nextIndex(trackedNode.index)];
}
```

Correctness Highlights: By INUSE, the tracked node is no longer in the collection if *redirect* is *null*. By the correctness of *skipRemovedElements*, when the tracked node is no longer in use, it is moved to the predecessor of the next element in the collection. Thus, in all cases, the aim is to return the next collection element, now from either *FORE_NODE*, or an node that is in use.

If *trackedNode* is *FORE_NODE* and the list is empty, or if *trackedNode* is the last element in the collection, then *AFT_NODE* is the correct element to return. If the element is *FORE_NODE* (but the collection is not empty), then the next element is that at the index corresponding to position 0. Otherwise, the element to return is the one at the next index. The rest of the correctness argument follows from PLACEMENT, and the correctness of *getIndex* and *nextIndex*. ■

The internal *prevCollectionElements* method returns a reference to the previous node in the collection. More specifically, if *trackedNode* is either at position p or if *trackedNode* is between positions $p-1$ and p , then a reference to the node at position $p-1$ is returned. If *trackedNode* is *AFT_NODE* then the node at position $size-1$ is returned (unless the collection is empty in which case *FORE_NODE* should be returned). It requires *trackedNode* is not at *FORE_NODE*. For the sake of efficiency, *skipRemovedElements* is used when *trackedNode* is no longer in the collection.

```
Node<E> prevCollectionElement() {
    if ((trackedNode == AFT_NODE && size == 0) ||
        (trackedNode.redirect == null && trackedNode.index == getIndex(0)))
        return FORE_NODE;
    if (trackedNode == AFT_NODE)
        return (Node<E>) a[getIndex(size-1)];
    if (trackedNode.redirect == null)
        return (Node<E>) a[prevIndex(trackedNode.index)];
    else
        return skipRemovedElements(trackedNode);
}
```

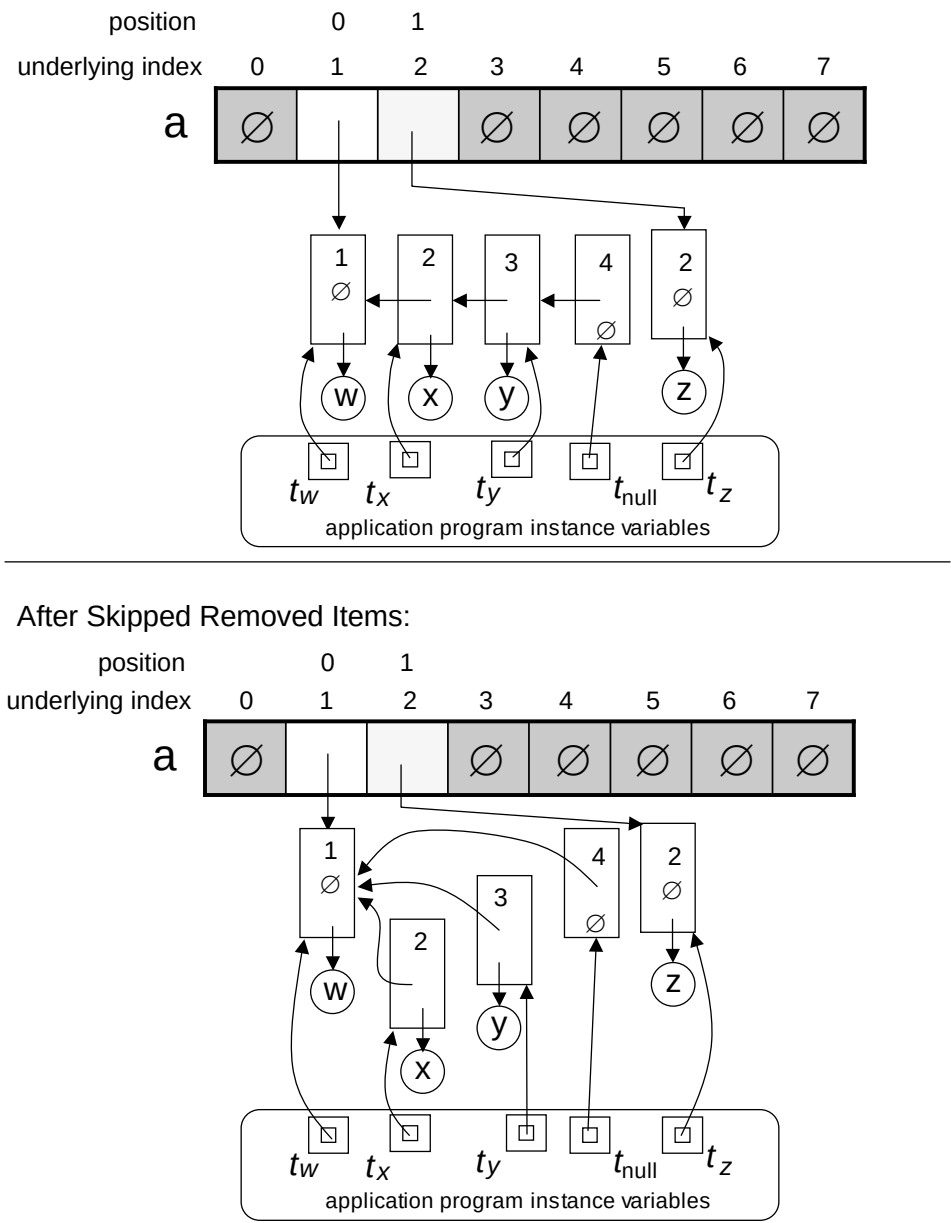


Figure 14.3

The updates made by *skipRemovedElements*(*t_{null}*).

Correctness Highlights: If *trackedNode* is *FORE_NODE* and the list is empty, or if *trackedNode* is the last element in the collection, then *AFT_NODE* is the correct element to return. Otherwise, if the element is *FORE_NODE* (but the collection is not empty), then the next element is that at the index corresponding to position *size* – 1.

By *INUSE*, if *redirect* is not *null*, then the tracked node is in use. Thus the desired element is that at the previous index. By the correctness of *skipRemovedElements*, when the tracked node is no longer in use, it is moved to previous element at the end of the redirect chain. The rest of the correctness argument follows from *PLACEMENT*, and the correctness of *getIndex* and *prevIndex*. ■

Recall the *advance* method moves the tracker to the next position, and returns *true* if and only if the tracker has not reached *AFT*. It throws an *AtBoundaryException* when the tracker is at *AFT*, since there is no place to advance.

```
public boolean advance() throws ConcurrentModificationException {
    if (trackedNode == AFT_NODE)
        throw new AtBoundaryException("Already at end.");
    checkValidity(); //throw ConcurrentModificationException if tracker invalidated
    trackedNode = nextCollectionElement();
    return trackedNode != AFT_NODE;
}
```

Correctness Highlights: Assuming the tracker is not currently at *AFT_NODE*, the tracker instance variable, *trackedNode*, is updated to the value returned by *nextCollectionElement* and the correctness follows from that of *nextCollectionElement*. The return value is correct since if *trackedNode* is at *AFT_NODE* after the update, then it has advanced past the end of the collection, and otherwise it has not. ■

Recall the *retreat* method moves the tracker to the previous position. It returns *true* if and only if the tracker has not reached *FORE*. It throws an *AtBoundaryException* when the tracker is at *FORE*, since then there is no place to retreat.

```
public boolean retreat() throws ConcurrentModificationException {
    if (trackedNode == FORE_NODE)
        throw new AtBoundaryException("Already before front.");
    checkValidity(); //throw ConcurrentModificationException if tracker invalidated
    trackedNode = prevCollectionElement();
    return trackedNode != FORE_NODE;
}
```

Correctness Highlights: The correctness argument is like that for *advance* except it depends on the correctness of *prevCollectionElement*. ■

The *hasNext* method returns *true* if there is some element after the current tracker position.

```
public boolean hasNext() throws ConcurrentModificationException {
    checkValidity(); //throw ConcurrentModificationException if tracker invalidated
    return (trackedNode != AFT_NODE && nextCollectionElement() != AFT_NODE);
}
```

Correctness Highlights: Follows by that of *nextCollectionElement* along with the observation that it also does not have a next element if *trackedNode* is already at *AFT_NODE*. ■

The tracker class provides two mutators: one that inserts a new element after the tracker and one that removes the element (if any) at the tracker. We first describe the *addAfter* method that takes *e*, the element to be added. It adds to the collection after the tracker location and returns a tracker for the new element. It throws an *AtBoundaryException* when the tracker is at *AFT*. If the tracker is between positions *p-1*, and *p* then the new element is inserted at position *p*.

```
public PositionalCollectionLocator<E> addAfter(E e) {
    int pos; //position to add new element
    if (trackedNode.redirect == null)
        pos = getPosition(trackedNode.index)+1;
    else {
        Node t = prevCollectionElement();
        if (t == FORE_NODE)
            pos = 0;
        else
            pos = getPosition(t.index) + 1;
    }
    return addImpl(pos, e);
}
```

Correctness Highlights: The correctness follows from that of *add*, *nextCollectionElement*, *INUSE*, and *SELFREFERENCE*. ■

The tracker *remove* method removes the element at the tracker and updates the tracker to be between its current position and the one before it. It throws a *NoSuchElementException* when the tracker is not at a valid position in the collection.

```
public void remove() {
    if (!inCollection())
        throw new NoSuchElementException();
    TrackedArray.this.remove(getPosition(trackedNode.index));
}
```

Correctness Highlights: By the correctness of *inCollection*, the exception is properly thrown. Otherwise, the tracked element is removed (or an exception is thrown) using the superclass *remove* method. The rest of the correctness follows from that of the superclass *remove* and *getPosition*. ■

14.7 Performance Analysis

We compare the time and space complexity to that of a circular dynamic array. We begin by considering the additional space usage for both the positional collection, and each tracker maintained

by the user program. The space usage for maintaining the collection with a dynamic circular array is the capacity of the underlying array. For a tracked array, each cell of the underlying array references a node. Since a node has three instance variables, the space used by the tracked array (excluding the element itself) is roughly four times that required of a circular array. As discussed under optimizations, this could be reduced to three times that of a circular array, by removing *redirect* if iteration need not be continued after a mutation occurs. Each tracker created (either via *addTracked*, *addAfter*, *iterator*, *iteratorAtEnd*, or *iteratorAt*), just stores a single reference to the node in the array.

Asymptotically, the time complexity for a tracked array is the same as that of a dynamic circular array (which is the same as that for a circular array). However, there is some low-order overhead incurred. First, whenever any array elements are moved or swapped, then the index stored within the tracker must be updated and this takes constant time per array element moved. This computation is whatever is required to enable the tracker to track an element and is fast in comparison to the rest of the computation performed. The other additional cost is incurred by *skipRemovedElements*. However, as discussed on page 209, *skipRemovedElements* has constant amortized cost.

In summary, the asymptotic time complexities of all methods are given in Table 12.4. For the locator methods, the asymptotic time complexities are the same as for the *Array* as shown in Table 11.9.

14.8 Quick Method Reference

TrackedArray Public Methods

- p. 199 *TrackedArray*()
- p. 199 *TrackedArray*(int capacity)
- p. 198 *TrackedArray*(int capacity, Comparator<? super E> equivalenceTester)
- p. 203 PositionalCollectionLocator<E> **addImpl**(int p, Object element)
- p. 203 PositionalCollectionLocator<E> **addTracked**(E value)
- p. 203 PositionalCollectionLocator<E> **addTracked**(int p, E value)
- p. 207 void **bucketSort**(Bucketizer <? super E> bucketizer)
- p. 204 void **clear**()
- p. 199 E **get**(int p)
- p. 205 PositionalCollectionLocator<E> **getLocator**(E value)
- p. 206 void **heapSort**(Comparator <? super E> comp)
- p. 206 void **insertionSort**(Comparator <? super E> comp)
- p. 204 PositionalCollectionLocator<E> **iterator**()
- p. 205 PositionalCollectionLocator<E> **iteratorAt**(int pos)
- p. 205 PositionalCollectionLocator<E> **iteratorAtEnd**()
- p. 206 void **mergeSort**(Comparator <? super E> comp)
- p. 206 void **quickSort**(Comparator <? super E> comp)
- p. 207 void **radixSort**(Digitizer <? super E> digitizer)
- p. 204 void **removeRange**(int fromPos, int toPos)
- p. 206 E **repositionElementByRank**(int i, Comparator <? super E> comp)
- p. 201 E **set**(int p, E element)
- p. 206 void **treeSort**(Comparator <? super E> comp)

TrackedArray Internal Methods

- p. 206 void **buildPriorityQueue**(PriorityQueue<Object> pq)
- p. 200 int **findPosition**(E element)
- p. 207 Bucketizer **getNodeBucketizer**(Bucketizer<? super E> bucketizer)
- p. 206 Digitizer **getNodeDigitizer**(Digitizer<? super E> digitizer)

- p. 205 Comparator **getSorter**(Comparator<? super E> comp)
- p. 202 void **move**(int fromPos, int toPos)
- p. 202 void **put**(int p, Object x)
- p. 199 E **readElement**(int p)
- p. 201 void **resizeArray**(int desiredCapacity)
- p. 200 void **shiftLeft**(int p1, int p2, int num)
- p. 201 void **shiftRight**(int p1, int p2, int num)
- p. 202 void **swapImpl**(int pos1, int pos2)
- p. 200 void **updateNodes**(int i, int num)
- p. 199 E **writeElement**(int p, E element)

TrackedArray.Tracker Public Methods

- p. 213 PositionalCollectionLocator<E> **addAfter**(E e)
- p. 212 boolean **advance**()
- p. 208 E **get**()
- p. 209 int **getCurrentPosition**()
- p. 212 boolean **hasNext**()
- p. 101 void **ignoreConcurrentModifications**(boolean ignore)
- p. 101 void **ignorePriorConcurrentModifications**()
- p. 208 boolean **inCollection**()
- p. 101 E **next**()
- p. 213 void **remove**()
- p. 212 boolean **retreat**()
- p. 208 E **set**(E element)

TrackedArray.Tracker Internal Methods

- p. 208 *Tracker*(Node<E> toTrack)
- p. 101 void **checkValidity**()
- p. 210 Node<E> **nextCollectionElement**()
- p. 210 Node<E> **prevCollectionElement**()
- p. 209 Node<E> **skipRemovedElements**(Node<E> ptr)
- p. 101 void **updateVersion**()

Chapter 15

Singly Linked List Data Structure

package `collection.positional`

`AbstractCollection<E>` implements *Collection<E>*

↳ `SinglyLinkedList<E>` implements *PositionalCollection<E>*, *Tracked<E>*

Uses: Java references

Used By: Stack (Chapter 19)

Strengths: A singly linked list supports constant time methods to add or remove an element after any element in the list whose location is already known. It can also efficiently add to the front or back of the list. It also supports a tracker that robustly tracks an element in the collection. A singly linked list tracker supports constant time operations to add after its tracked element, peek at the next element, and remove the next element. The space usage is moderate with roughly 2 references per element.

Weaknesses: A singly linked list does not support constant time access via position. To access the element at a given position p takes $O(p + 1)$ time. Similarly, the locator method *getCurrentPosition* takes time $O(p + 1)$ if the element tracked is at position p . Because a singly linked list does not include a previous pointer for each list item, locating the list item just before a given list requires traversing the list from the front until reaching the desired list item. Therefore, iterating backward through the list takes quadratic time. Similarly, to remove an element in position p of the collection without a tracker takes time $O(p + 1)$, since the next pointer in the previous item must be changed.

Critical Mutators: *heapsort*, *insertionsort*, *mergesort*, *quicksort*, *radixsort*, *repositionElement*, *ByRank*, *swap*, *treesort*

Competing Data Structures: A doubly linked list (Chapter 16) is a better choice if removing a tracked element is a frequent operation. Also if it is important to quickly locate elements near the end of the collection or to efficiently iterate through the collection in reverse order, the additional space used by a doubly linked list data structure is justified.

An array-based data structure is a better choice if it is important to be able to efficiently access elements by position, and if it is acceptable for add and remove by position to take linear time for positions near the middle of the collection. An array-based data structure is also preferable when it is important to determine the position of a tracked element in constant time.

15.1 Internal Representation

We now describe the internal representation for a singly linked list, which is composed of list items.

Instance Variables and Constants: Variable *size* is inherited. To eliminate special cases in the code for treatment of the first element, each singly linked list has a dummy list item *head* that serves as a **sentinel head**. The *head* sentinel is also used by the tracker as *FORE*. The sentinel introduces very little space overhead (and no overhead in computation time). To facilitate the easy extension of a singly linked list to a doubly linked list, which uses a sentinel tail, we logically treat a singly linked list as ending at “tail” which is physically just *null*. Finally, to support constant time insertion to the end of the list, the instance variable *last* always references the last item in the list, which is *FORE* when the list is empty.

```
protected ListItem<E> head;           //sentinel head ListItem
private ListItem<E> last;             //reference to last item
```

Sometimes a tracker will refer to a list item that has been removed from the list. To distinguish these list items from others, we replace their associated data by the singleton constant *REMOVED*. This is why the list item’s *data* field is of type *Object* instead of type *E*.

```
static final Object REMOVED = new Object(); //singleton for a removed item
```

Terminology: Since a singly linked list is a positional collection, it is convenient to associate position numbers with each list item. We introduce several definitions that relate the underlying representation to the user view of the positional collection.

- We let x_p denote the p^{th} list item in the collection, where x_{-1} refers to the head sentinel. More formally, we define $x_{-1}, \dots, x_{n-1}$ recursively, as

$$x_p = \begin{cases} \text{head} & \text{for } p = -1 \\ x_{p-1}.\text{next} & \text{for } p = \{0, \dots, n-1\} \end{cases}$$

- We say that list items $x_{-1}, x_0, \dots, x_{n-1}$ are **reachable** when they can be accessed from *head*. These list items hold the elements in the collection. We say that all other list items are **unreachable**. (Note that an unreachable list item might be accessible from a Tracker, but it is not accessible by following the *next* pointers starting at *head*.)
- Let x be a reference to a list item. We define $\text{pos}(x)$ recursively as follows:

$$\text{pos}(x) = \begin{cases} \text{size} & \text{if } x = \text{tail} \\ \text{pos}(x.\text{next}) & \text{if } x.\text{data} = \text{removed} \\ \text{pos}(x.\text{next}) - 1 & \text{otherwise} \end{cases}$$

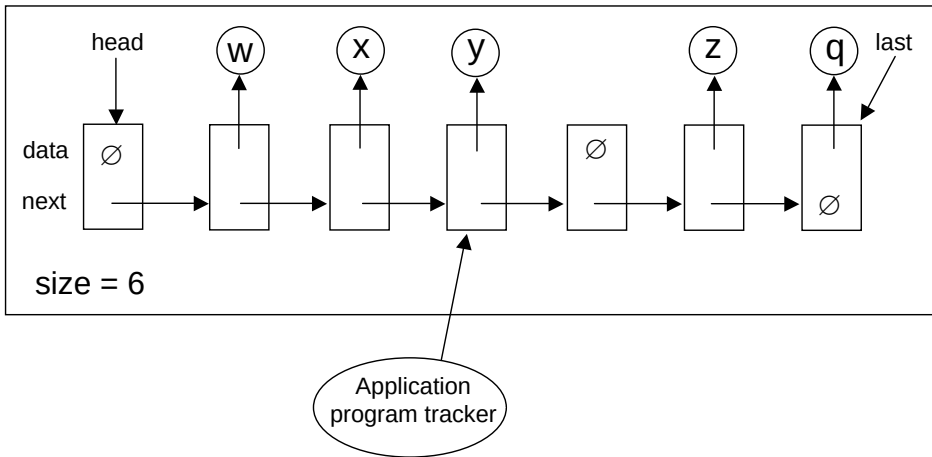
For a reachable *ListItem* x_p , note that $\text{pos}(x_p) = p$. We will think of an unreachable list item x as being logically between positions $\text{pos}(x) - 1$ and $\text{pos}(x)$.

- We define the **last list item** to be $x_{\text{size}-1}$.

Abstraction Function: Let L be a singly linked list. The abstraction function

$$AF(L) = \langle u_0, u_1, \dots, u_{\text{size}-1} \rangle \text{ such that } u_p = x_p.\text{data}.$$

Populated Example: A singly linked list for the positional collection $\langle w, x, y, \text{null}, z, q \rangle$, is illustrated in [Figure 15.1](#). This figure shows a single tracker for the element at position 2 (that has data value y). [Figure 15.2](#) shows the same collection after y has been removed.

**Figure 15.1**

A populated example of a singly linked list for the positional collection $\langle w, x, y, \text{null}, z \rangle$. Each list item is shown as a rectangle. While we show the list items in a straight line, they are really spread throughout memory, not necessarily in that order. When a new element is inserted, the existing list items do not move.

Design Notes: To support the use of a sentinel tail in inherited methods in `DoublyLinkedList`, we introduced a method `getTail` that here always returns `null`. To save the cost of that method call, one could create a separate class (not extended by `DoublyLinkedList`) in which all of the calls to `getTail` were replaced by the value `null`. The redirect chain makes use of both the **proxy design pattern**, and the idea of path compression that originated in the union-find data structure (Section 6.3).

Optimizations: In our implementation, we have chosen to use a sentinel for the head. While this introduces the small space overhead of a single list item, it avoids the need for special cases in the code when adding or removing an element from the front of the list. While this space overhead is usually negligible, it can be unacceptable if the lists are extremely small. For example in [Chapter 23](#) an array of singly linked lists is needed where on average each list contains a single element. Also a tracker is not needed for that application. Using a singly linked list in that case would more than triple the space usage, which is excessive. While we could rewrite `SinglyLinkedList` to minimize space usage, doing so would make the code slightly less efficient because of the conditional that would be needed to check if an element is being inserted to the front of the list. Thus, we have chosen to include in [Chapter 23](#) a direct implementation of a space efficient singly linked list that does not support a tracker.

To improve efficiency, `last` is directly accessed and updated by any methods that are overridden by the `DoublyLinkedList` class. Alternatively, `getLast` and `setLast` could be used.

The `clear` method takes linear time. Since there could be a tracker positioned at any given node, each node in the list must be marked as deleted. If an untracked implementation is all that is needed, `clear` can be implemented in constant time by just setting `head` to `null`, and the size to 0.

The sorting algorithms could be implemented by first moving the list items into an array, and then using the same ideas illustrated in the `Tracked Array` class ([Chapter 14](#)) to wrap a comparator, digitizer, or bucketizer, and then use the `Array` class sorting algorithms. However, with the exception of heap sort and tree sort, we have provided direct in-place implementations to improve both the space efficiency and time complexity of these methods.

The implementation of quicksort presented in this chapter applies a standard optimization [23] of only applying the recursive method to a specified cut-off, and then use insertion sort to complete the sort.

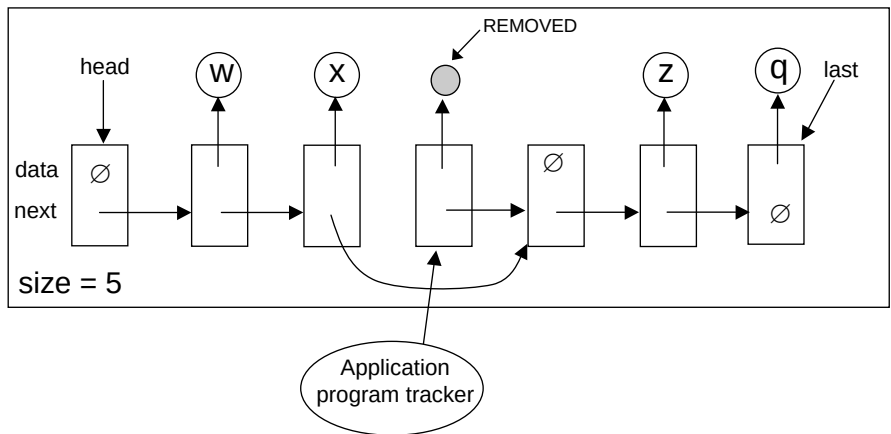


Figure 15.2
A populated example of the singly linked list from Figure 15.1 after y has been removed. The gray object is the singleton object that marks a list item as having been removed.

15.2 Representation Properties

We inherit representation property SIZE and introduce four additional representation properties. Recall that we consider the head sentinel to be at position -1 and the tail to be at position $size$.

PLACEMENT: The element at user position p is held in the p^{th} list item. More formally, for $p \in \{0, \dots, size-1\}$, $x_p.data = u_p$.

ENDSATTAIL: The *next* field for the last item in the list (which by PLACEMENT is that for element $size - 1$) is *tail*. That is, $x_{size-1}.next = tail$.

REMOVED: For any unreachable list item x , any locator tracking x is logically between $pos(x)-1$ and $pos(x)$, and $x.data = REMOVED$.

LAST: The element referenced by *last.data* is the last element in the collection. When the collection is empty, *last* references *head* sentinel.

ENDSATTAIL is used to guarantee that a search to locate an element always ends. REMOVED is used to argue that the trackers have the specified behavior when elements are removed.

15.3 List Item Inner Class

ListItem $\langle E \rangle$

A singly linked list is a pointer-based structure composed of list items, where each list item holds an element and a reference to the next list item (or to tail for the last element).

```
Object data;
ListItem<E> next;
```

The `ListItem` constructor takes *data*, the element held in the list item. It sets the data to the given value.

```
ListItem(Object data) {this.data = data;}
```

The `setNext` method takes *next*, the value to update the *next* reference. Most of the differences between a singly linked list and a doubly linked list can be handled simply by overriding this method.

```
protected void setNext(ListItem<E> next) {this.next = next;}
```

The `moveNextAfter` method takes *destination*, a reference to a list item. It moves the list item after this one so that it immediately follows the destination. This method must not be called on the last element in the collection. If *destination* is this list item, then nothing happens because it makes no sense to move an item after itself. Similarly, if *destination* is *this.next*, the method returns without modifying the list.

```
protected boolean moveNextAfter(ListItem<E> destination) {
    if (destination != next && destination != this) {
        ListItem<E> temp = next;
        this.setNext(temp.next);
        temp.setNext(destination.next);
        destination.setNext(temp);
        return true;
    }
    return false;
}
```

15.4 Singly Linked List Methods

In this section we present internal and public methods for the `SinglyLinkedList` class.

15.4.1 Constructors and Factory Methods

The constructor takes no arguments and creates an empty collection.

```
public SinglyLinkedList() {
    initialize();
}
```

To prevent the need to change methods in the doubly linked list to allocate the correct list item type, we use the **factory method design pattern** (Section C.9) to generate a new list item. Specifically, the `newListItem` method takes *value*, the element to insert, and returns a new list item with the given value.

```
ListItem<E> newListItem(E value) {
    return new ListItem<E>(value);
}
```

The *initialize* method updates this collection to be empty.

```
void initialize(){
    head = newListItem(null);
    head.setNext(getTail());
    last = head;
    size = 0;
}
```

Correctness Highlights: SIZE is satisfied since $size = n = 0$. The collection holds no elements, so PLACEMENT vacuously holds. Since *head* is initialized to *getTail()*, ENDSATAIL holds. Finally, there are no unreachable list items created by this method, so REMOVED vacuously holds. LAST also holds since it should reference the *head* sentinel when the collection is empty. ■

15.4.2 Representation Accessors

Although our singly linked list implementation does not use a tail sentinel, it is helpful to have a method that returns the value of “tail.” Then DoublyLinkedList can override this method to return the tail sentinel. Here we can avoid the extra space of a tail sentinel by simply returning *null*.

```
ListItem<E> getTail() {
    return null;
}
```

The *getLast* method returns a reference to the list item for the last element in the collection.

```
ListItem<E> getLast() {
    return last;
}
```

15.4.3 Algorithmic Accessors

The *getPtrForPrevElement* method takes *ptr*, a reference to a list item, and returns a reference to the previous list item. It requires that *ptr* references a reachable list item. Since there is no previous pointer in a singly linked list, this method takes linear time, in the worst case.

```
ListItem<E> getPtrForPrevElement(ListItem<E> ptr){
    ListItem<E> loc = head;
    while (loc != getTail() && loc.next != ptr)
        loc = loc.next;
    return loc;
}
```

Correctness Highlights: By PLACEMENT and the fact that $x_0, \dots, x_{size-1}$ are accessible from *head*, all list items in the collection are accessed. Termination is guaranteed by ENDSATAIL, and the requirement that *ptr* references a reachable list item. ■

There is no efficient mechanism to find the element in a linked list at a given position. However, this functionality is required by several of the public methods. The *getPtr* method takes *p*, a valid user position or -1, and returns a reference to the element in the collection at position *p* or the head sentinel if *p* = -1. It requires that $-1 \leq p < \text{size}$.

```
protected ListItem<E> getPtr(int p){
    if (p == size - 1)                                //special case to efficiently retrieve the last item
        return getLast();
    ListItem<E> current = head;
    for (int i = 0; i ≤ p; i++)
        current = current.next;
    return current;
}
```

Correctness Highlights: Follows by PLACEMENT, the requirement that *p* is an integer between -1 and *size*-1, and by the definition that $x_{-1} = \text{head}$. Finally, when *p* is *size* - 1 the correctness follows from LAST. ■

The public *get* method takes *p*, a valid user position, and returns the object at the position *p*. It throws a *PositionOutOfBoundsException* when the *p* is not a valid position.

```
public E get(int p) {
    if (p < 0 || p ≥ size)
        throw new PositionOutOfBoundsException(p);
    return (E) getPtr(p).data;
}
```

Correctness Highlights: Follows from the correctness of *getPtr(p)* and PLACEMENT. ■

The next two methods are used to locate a desired element. Since there is not an efficient method to move backwards in a singly linked list, for the sake of efficiency we instead return the list item that precedes the one with the desired element. The *getPtrForPrevElement* method takes *value*, the target, and returns a reference to the first reachable list item that immediately precedes one holding an element equivalent to *value*. The method returns *null*, if there is no equivalent element.

```
protected ListItem<E> getPtrForPrevElement(E value) {
    ListItem<E> loc = head;
    while (loc.next != getTail()) {
        if (equivalent(value, (E) loc.next.data))
            return loc;
        loc = loc.next;
    }
    return null;
}
```

Correctness Highlights: Follows by the correctness of *equivalent* and PLACEMENT. Termination is guaranteed by ENDSATAIL. ■

Recall that the method *contains* takes *value*, the target, and returns *true* if and only if an element equivalent to *value* exists in the collection.

```
public boolean contains(E value) {
    return getPtrForPrevElement(value) != null;
}
```

The *positionOf* method takes *value*, the target, and returns the position in the collection for the first occurrence (if any) of an element equivalent to *value*. It throws a *NoSuchElementException* when no element in the collection is equivalent to *value*.

```
public int positionOf(E value) {
    ListItem<E> ptr;                                //reference to ListItem currently at in traversal
    int pos;                                          //position in the collection of ptr.data
    for (pos = 0, ptr = head.next; ptr != getTail(); pos++, ptr = ptr.next)
        if (equivalent(value, (E) ptr.data))
            return pos;
    throw new NoSuchElementException();
}
```

Correctness Highlights: By ENDSATTAIL this will terminate if no equivalent element is in the collection. The rest of the correctness follows from PLACEMENT. ■

15.4.4 Representation Mutators

The *setLast* method takes *last*, the list element that is last in the collection.

```
protected void setLast(ListItem<E> last) {
    this.last = last;
}
```

Correctness Highlights: This method preserves LAST. ■

15.4.5 Content Mutators

Recall that *set* takes *p*, a valid user position, and *value*, the element to put at position *p*. It returns the prior element at position *p*. It throws a *PositionOutOfBoundsException* when *p* is not a valid position.

```
public E set(int p, E value) {
    if (p < 0 || p ≥ size)
        throw new PositionOutOfBoundsException(p);
    ListItem<E> ptr = getPtr(p);
    E oldData = (E) ptr.data;
    ptr.data = value;
    return oldData;
}
```

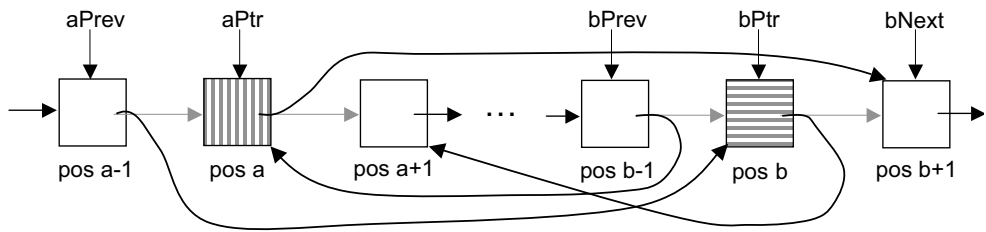
Correctness Highlights: By SIZE the exception is appropriately thrown. By the correctness of *getPtr(p)*, $ptr = x_p$. Thus, after this method executes, $x_p.data = u_p = value$, preserving PLACEMENT. ■

The *swapAfter* method takes *aPrev*, a reference to list item before the one to be swapped, and *bPrev*, a reference to the list item before the other one to be swapped. While it may seem more natural for the parameters to directly reference the list items to be swapped, in order to implement this method in constant time with a singly linked list, it is necessary to be provided with references to the list items that precede those to be swapped.

```
protected void swapAfter(ListItem<E> aPrev, ListItem<E> bPrev) {
    if (aPrev != bPrev) {
        ListItem<E> aPtr = aPrev.next;           //one list item to be swapped
        ListItem<E> aNext = aPtr.next;           //and the list item that follows it
        ListItem<E> bPtr = bPrev.next;           //other element to be swapped
        ListItem<E> bNext = bPtr.next;           //and the list item that follows it
        if (bPtr == aPrev)                        //special case if "a" immediately precedes "b"
            aPtr.setNext(bPtr);
        else {
            aPtr.setNext(bNext);
            aPrev.setNext(bPtr);
        }
        if (aPtr == bPrev)                        //special case if "b" immediately precedes "a"
            bPtr.setNext(aPtr);
        else {
            bPtr.setNext(aNext);
            bPrev.setNext(aPtr);
        }
        if (last == aPtr)                         //preserve last
            last = bPtr;
        else if (last == bPtr)
            last = aPtr;
        version.increment();                      //invalidate all active trackers as iterators
    }
}
```

Correctness Highlights: From the correctness of *getPtr* and PLACEMENT, *aPrev* points to the list item for position *a-1*, *aPtr* points to the list item for position *a*, *bPrev* points to the list item for position *b-1*, *bPtr* points to the list item for position *b*, and *bNext* points to the list item for position *b+1*. As illustrated in Figure 15.3, the final four statements adjust the *next* pointers so that the list item that was in position *b* (reference by *bPtr*) is preceded by *aPrev* and followed by the list item holding the element at position *a+1*. Similarly, the list item that was in position *a* is moved into position *b*. Thus PLACEMENT is maintained.

The last list item in the collection changes only when either *aPrev* or *bPrev* is equal to *last*. In these cases, the other becomes the last list item in the collection. Thus LAST is preserved. None of the other properties are affected. Finally, all trackers must be invalidated when this method is executed because the swap may cause some persistent element to be seen twice or not at all. ■



Equivalently:

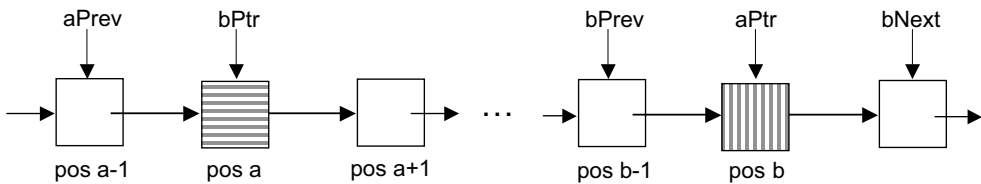


Figure 15.3

An illustration of the *swapAfter* and *swap* methods. Only the *next* pointers in the relevant portion of the *SinglyLinkedList* instance are shown. In the top figure the original values of the updated pointers are shown in light gray. In the bottom figure only the updated pointer values are shown and the squares are rearranged to be in their logical order.

The method *swap* takes *a*, a valid position, and *b*, a valid position. It swaps the elements held in positions *a* and *b*. It throws a *PositionOutOfBoundsException* when either *a* or *b* is not a valid position.

```
public void swap(int a, int b) {
    if (a < 0 || a ≥ size || b < 0 || b ≥ size)
        throw new PositionOutOfBoundsException();
    ListItem<E> aPrev = getPtr(a-1);
    ListItem<E> bPrev = getPtr(b-1);
    swapAfter(aPrev, bPrev);
}
```

Methods to Perform Insertion

We now introduce an internal method that is used to perform all insertions. The public insertion methods will perform any required checks for exception handling and call this method with the appropriate parameters. The *insertAfter* method takes *ptr*, a reference to a list item in the collection (possibly the sentinel head), and *value*, the element to insert. This method inserts a list item holding the new element after the list item referenced by *ptr*. It returns a tracker to the new element.

```
protected PositionalCollectionLocator<E> insertAfter(ListItem<E> ptr, E value) {
    ListItem<E> newItem = newListItem(value);
    newItem.setNext(ptr.next);
    ptr.setNext(newItem);
    if (last == ptr)                                     //preserve Last
```



```

        last = newItem;
        size++;
        return new Tracker(newItem);
    }
    //preserve Size

```

Correctness Highlights: Let $p = \text{pos}(\text{ptr})$. By PLACEMENT, $\text{ptr.data} = u_p$ and $\text{ptr.next.data} = x_{p+1}.\text{data} = u_{p+1}$. It is easily verified (using a diagram like that of Figure 15.3) that x is placed at position $p+1$ since x is placed after ptr and before x_{p+1} . Observe that by inserting x in this position, implicitly the positions for $u_{p+1}, \dots, u_{\text{size}-1}$ are incremented by 1. Thus, once value is added after position p (i.e., in position $p+1$), the resulting abstraction is

$$\langle u_0, u_1, \dots, u_p, \text{value}, u_{p+1}, \dots, u_{\text{size}-1} \rangle = \langle u'_0, u'_1, \dots, u'_p, \text{value}, u'_{p+2}, \dots, u'_{\text{size}'-1} \rangle.$$

We use a proof by cases to argue that ENDSATTAIL is preserved. Clearly ENDSATTAIL is preserved when $p \in \{-1, \dots, \text{size}-2\}$ since $x_{\text{size}-1}$ is not changed. Suppose $p = \text{size} - 1$. By ENDSATTAIL, $x_{p+1} = \text{getTail}()$. When *insertAfter* completes, $x_{\text{size}'-1} = x$, thus preserving ENDSATTAIL.

If ptr was the last list item, the new item will now be the last one. The conditional checks for this case to preserve LAST. Since $\text{size}' = \text{size} + 1$, PLACEMENT is preserved. Finally REMOVED holds since the *ListItem* holds the data element and is reachable. ■

Recall that *add* takes p , the position where the element is to be placed, and value , the element to insert. It inserts value at position p and increments the position number for the elements that were at positions $p, \dots, \text{size}-1$. It throws a *PositionOutOfBoundsException* when p is neither size nor a valid position.

```

public void add(int p, E value) {
    if (p < 0 || p > size)
        throw new PositionOutOfBoundsException(p);
    insertAfter(getPtr(p-1), value);
}

```

Correctness Highlights: By SIZE, an exception is appropriately thrown. The new element should be inserted immediately after the element at position $p-1$. (Recall that the head sentinel is the element at position -1 which is a valid input for *getPtr*.) The correctness thus follows from that of *getPtr* and *insertAfter*. ■

Finally, the non-indexed *add* method takes value , the new element, and inserts it at the end of the positional collection.

```

public void add(E value) {
    insertAfter(getLast(), value);
}

```

Correctness Highlights: Follows from LAST and the correctness of *insertAfter*. ■

Similarly, the *addTracked* method takes value , the new element and inserts value into the collection at an arbitrary position. It returns a tracker that tracks the new element.

```

public Locator<E> addTracked(E value) {
    return insertAfter(getLast(), value);
}

```

Correctness Highlights: Also follows from LAST and the correctness of *insertAfter*. ■

Methods to Perform Deletion

Observe that the positions of each list item (and their respective elements) are defined implicitly based on the list structure. Thus, when an element is added to the list, the positions of all elements after the new element implicitly increase by one without any additional computation. Likewise, when an element is removed from the list, the positions of all elements after it implicitly are reduced by one.

As with *insertAfter*, we introduce an internal method *removeNext* that takes *ptr*, a reference to the list item (possibly the sentinel head item) preceding the one to remove. It returns the removed element. This method requires that *ptr* references *head* or a reachable list item. It throws a *NoSuchElementException* when *ptr* references the last list item.

```

protected E removeNext(ListItem<E> ptr) {
    if (ptr.next == getTail())
        throw new NoSuchElementException();
    if (last == ptr.next)                                     //preserve Last
        last = ptr;
    ListItem<E> x = ptr.next;
    ptr.setNext(x.next);                                     //remove the list item after ptr
    size--;                                                  //preserve Size
    E result = (E) x.data;
    x.data = REMOVED;                                       //preserve Removed
    return result;
}

```

Correctness Highlights: Let p denote the position of the element that is to be removed, which is well-defined by the requirement of this method). By ENDSATTAIL if *ptr.next* references the tail, then *ptr* references the last list item. In this case there is nothing after it to remove, so the exception is appropriately thrown. Clearly all properties are preserved when no update occurs.

Otherwise, $p \in \{0, 1, \dots, \text{size} - 2\}$ and u_{p+1} is removed. By PLACEMENT, $y = x.\text{next}$, is at position $p+2$. SIZE is preserved since both *size* and *n* increase by one. We now argue that PLACEMENT is preserved. By PLACEMENT, before *removeNext* executes the user sequence is

$$\langle u_0, \dots, u_p, u_{p+1}, u_{p+2}, \dots, u_{\text{size}-1} \rangle = \langle u_0, \dots, \text{ptr.data}, x.\text{data}, y.\text{data}, \dots, u_{\text{size}-1} \rangle.$$

The update $\text{ptr.next} = x.\text{next}$ in combination with $\text{size}' = \text{size} - 1$ modifies the user sequence to:

$$\langle u_0, \dots, \text{ptr.data}, y.\text{data}, \dots, u_{\text{size}-1} \rangle = \langle u'_0, \dots, u'_p, u'_{p+1}, \dots, u'_{\text{size}'-1} \rangle$$

thus preserving PLACEMENT.

If x is not at position $\text{size} - 1$ then ENDSATTAIL is trivially preserved. We now consider when x is at position $\text{size} - 1$. In this case, by ENDSATTAIL, y is the tail, so the update of *ptr.next* to *x.next* preserves ENDSATTAIL.

Finally, we argue that `REMOVED` is preserved. Since the removed element was at position p , after it is removed any tracker to that element would be between positions $p-1$ and p . After this method has completed, the list item referenced by x will be unreachable, so by definition $pos(x) = pos(y)$. No change is made to y , so by `PLACEMENT` it references the object in the collection that was at position $p+1$ but after the remove will be at position p . Thus, upon completion of this method $pos(x) = p$ as the `REMOVED` property requires. Also $x.data$ is set to the value `REMOVED`. ■

Recall that `removeRange` takes as input `fromPos`, a valid position, and `toPos`, a valid position. This method requires $0 \leq fromPos \leq toPos < size$. It removes the elements at positions `fromPos`, $\dots$, `toPos`, inclusive, from the collection and decrements the positions of the elements at positions `toPos+1` to `size-1` by `toPos-fromPos+1` (the number of elements being removed). It throws a `PositionOutOfBoundsException` when either of the arguments is not a valid position, and it throws an `IllegalArgumentException` when `fromPos` is greater than `toPos`.

```
public void removeRange(int fromPos, int toPos) {
    if (fromPos < 0 || toPos ≥ size)
        throw new PositionOutOfBoundsException();
    if (fromPos > toPos)
        throw new IllegalArgumentException();
    ListItem<E> beforeFromPos = getPtr(fromPos-1);
    ListItem<E> ptr = beforeFromPos.next;
    for (int i = fromPos; i ≤ toPos; i++, ptr = ptr.next)
        ptr.data = REMOVED;                                //mark items as removed
    beforeFromPos.setNext(ptr);                               //remove them
    size = size - (toPos - fromPos + 1);                       //preserve Size
    if (last.data == REMOVED)                                  //preserve Last
        last = beforeFromPos;
}
```

Correctness Highlights: This method requires that $0 \leq fromPos < toPos \leq size - 1$ and will throw an illegal argument exception if this requirement does not hold. By the correctness of `getPtr`, the local variable `ptr` references the list item at position `fromPos-1`. Observe that the element that is at position `fromPos-1` at the start of this method, will still be at position `fromPos-1` when the method completes. Thus the for loop preserved `REMOVED` by marking the elements that were at positions `fromPos`, $\dots$, `toPos` as removed. Observe that `beforeFromPos` references the node at position `fromPos - 1`, and `ptr` references the node at position `toPos + 1`. So by the correctness of `setNext`, executing `beforeFromPos.setNext(ptr)` preserves `PLACEMENT` and `ENDSATTAIL`. Since `toPos - fromPos + 1` elements are removed, subtracting this quantity from the variables `size` preserves the `SIZE` property. when the method was called. Finally, the last conditional preserves `LAST`. ■

Recall, the `remove` method takes p , a valid position. It removes the element at position p and decrements the positions of $u_{p+1}, \dots, u_{size-1}$ by one. It returns the removed element u_p , or throws a `PositionOutOfBoundsException` when p is not a valid position.

```
public E remove(int p) {
    if (p < 0 || p ≥ size)
        throw new PositionOutOfBoundsException(p);
    ListItem<E> ptr = getPtr(p-1);
    return removeNext(ptr);
}
```

Correctness Highlights: The correctness follows from that of *getPtr* and *removeNext*. ■

The method *removeFirst* removes the element at position 0 and decrements the position for the elements that were at positions 1 to *size-1*. It returns the removed element, or throws a *NoSuchElementException* when the collection is empty.

```
public E removeFirst() {
    if (isEmpty())
        throw new NoSuchElementException("collection is empty");
    return removeNext(head);
}
```

Correctness Highlights: The correctness follows from that of *removeNext* and PLACEMENT which implies that *head.next.data* = *u₀*. ■

The method *removeLast* removes the element at position *size-1*, and returns the removed element. It throws a *NoSuchElementException* when the collection is empty.

```
public E removeLast() {
    if (isEmpty())
        throw new NoSuchElementException("collection is empty");
    return remove(size-1);
}
```

Correctness Highlights: By SIZE, the last element of the collection is that at position *size-1*. The rest of the correctness follows from that of *isEmpty* and *remove*. ■

The *remove* method from the *Collection* interface that takes *value*, the element to remove, removes the first element in the collection equivalent to *value*. It returns *true* if and only if an element is removed. Observe that if the user program has not maintained a tracker to this element, then the list item becomes garbage and is eligible to be reclaimed by Java's garbage collector.

```
public boolean remove(E value) {
    ListItem<E> ptr = getPtrForPrevElement(value);
    if (ptr == null)
        return false;
    removeNext(ptr);
    return true;
}
```

Correctness Highlights: Follows from that of *getPtrForPrevElement* and *removeNext*. ■

The *clear* method removes all elements from the collection. Although the inherited *clear* method would work (using the iterator to remove each item individually), we override it with a more efficient algorithm that marks every reachable list item as removed, and then reinitializes the list to be empty.

```

public void clear() {
    ListItem<E> tail = getTail();
    for (ListItem<E> ptr = head.next; ptr != tail; ptr = ptr.next)
        ptr.data = REMOVED;
    size = 0;
    head.setNext(tail);
    last = head;
}

```

Correctness Highlights: Setting *data* to *REMOVED* for all elements preserves *REMOVED*. *SIZE* is satisfied since *size* = *n* = 0. The collection holds no elements, so *PLACEMENT* vacuously holds. Since *head* is set to *getTail()*, *ENDSATAIL* holds. *LAST* also holds since it should reference the *head* sentinel when the collection is empty. ■

15.4.6 Locator Initializers

The *iterator* method creates a new marker that is at *FORE*.

```

public PositionalCollectionLocator<E> iterator() {
    return new Tracker(head);
}

```

Correctness Highlights: The correctness of the iterator follows from the correctness of the locator methods, and the fact that *head* serves the role of *FORE*. ■

The *iteratorAtEnd* method creates a new marker that is at *AFT*.

```

public PositionalCollectionLocator<E> iteratorAtEnd() {
    return new Tracker(getTail());
}

```

Correctness Highlights: The correctness of the iterator follows from the correctness of the locator methods and the fact that “tail” serves the role of *AFT*. ■

The *iteratorAt* method takes *pos*, the user position of an element, and returns a new marker that is at the given position. It throws a *NoSuchElementException* when the given position is not a valid user position.

```

public PositionalCollectionLocator<E> iteratorAt(int pos){
    ListItem<E> ptr = getPtr(pos);
    if (ptr == null)
        throw new NoSuchElementException();
    return new Tracker(ptr);
}

```

The method *getLocator* takes *element*, the element to track. It returns a tracker that has been initialized at the given element. As with the *iterator* method, this method also enables navigation. It throws a *NoSuchElementException* when the given element is not in the collection.

```

public PositionalCollectionLocator<E> getLocator(E element){
    ListItem<E> ptr = getPtrForPrevElement(element);
    if (ptr == null)
        throw new NoSuchElementException();
    return new Tracker(ptr.next);
}

```

15.5 Sorting Algorithms Revisited

The simplest approach for sorting using a list-based implementation would be to add all elements into an array, and then sort that array. However, such an approach would not preserve the trackers which reference the list items. One could preserve the trackers by using an approach similar to the one used for the tracked array (Chapter 14) in which all list items are inserted into any array and then sorted using a comparator defined over list items, and then the list could be rebuilt.

However, most of the sorting algorithms presented in Section 11.4, can be efficiently implemented directly on a list. In particular, we provide implementations of insertion sort, mergesort, and quicksort that work in-place on the list. Similarly, radix sort and bucket sort can be implemented very efficiently directly on the list. The asymptotic time complexities of all the sorting algorithms are the same as those presented in Section 11.4. Also, with the exception of using a different partition method for quicksort, the fundamental algorithms and their correctness arguments do not change. Therefore, we do not repeat the correctness arguments or analysis here.

The implementations of tree sort and heap sort use a similar approach to the one we introduced for the tracked array (Chapter 14) in which we define a comparator over list items, and then place the list items (as opposed to the elements) into the heap or tree. To support rebuilding the list, we introduce the *addItemLast* method that takes *x*, the item to be placed at the end of the list.

```

protected void addItemLast(ListItem<E> x) {
    x.setNext(getTail());
    last.setNext(x);
    last = x;
}

```

//preserves EndsAtTail
//places x after last item
//preserves Last

15.5.1 Insertion Sort

At a high-level, our list-based insertion sort implementation is like that discussed in Section 11.4.1. Consider when element *x* in position *p* is being placed into the correct position into the sorted subcollection from positions 0 to *p* − 1. Unlike in an array in which the elements greater than *x* must be moved to the right to make space to insert *x*, with a linked list we need just find the correct position to insert *x* and then move it there. To insert *x* into its proper position, we need access to both the element that was before *x* (the one in position *p* − 1) and the element that will precede *x* in its final position. As with the array-based implementation, this sort is stable and in-place. To ensure it is stable, we continue moving *ptr* forward in the list until the item that follows it is strictly greater than *x*, ensuring that *x* is placed after any equal elements that had preceded it in the collection. The *insertionsort* method with no arguments uses the default comparator to order the elements.

```

public void insertionsort() {
    insertionsort(Objects.DEFAULT_COMPARATOR);
}

```

The *insertionsort* method with an argument takes *comp*, the comparator to use to order the elements.

```

public void insertionsort(Comparator<? super E> comp) {
    if (getSize() > 1) {
        ListItem<E> loc = head.next;
        while (loc.next != getTail()) {
            ListItem<E> ptr = head;
            while (ptr != loc && comp.compare((E) ptr.next.data, (E) loc.next.data) ≤ 0)
                ptr = ptr.next;
            if (ptr != loc) {
                if (loc.next == getLast())
                    setLast(loc);
                loc.moveNextAfter(ptr);
            }
            else
                loc = loc.next;
        }
        version.increment();
    }
}

```

//loc.next is placed into position
//find ptr after which loc.next should be placed
//if ptr.next.data ≤ loc.next.data, need to move forward
//loc.next needs to be moved after ptr
//preserves Last
//after moving loc.next, loc is last
//move loc.next into place
//loc.next is already in place
//continue to the next list item to process
//invalidate active trackers for iteration

Correctness Highlights: By ENDSATTAIL and the correctness of *getTail*, all elements are processed. The correctness argument for insertion sort itself is like that in the Array class. By the correctness of *moveNextAfter*, each element is placed in the desired position in a way to preserve PLACEMENT and ENDSATTAIL. LAST is preserved by reassigning *last* whenever the element currently last is moved earlier in the collection. Finally, all trackers must be invalidated when this method is executed because the call to *moveNextAfter* may cause some persistent element to be seen twice or not at all. ■

15.5.2 Mergesort

In this section we present our list-based implementation of mergesort. Unlike an array-based implementation, with a linked list the merge method can be performed in place. Recall that the *mergesort* method with no arguments uses the default comparator to order the elements.

```

public void mergesort() {
    mergesort(Objects.DEFAULT_COMPARATOR);
}

```

The *mergesort* method that takes *comp*, the comparator to use to order the elements, sorts the list using mergesort. This method simply calls the recursive mergesort implementation on the entire list and uses its return value to reset *head.next* to the first item in the sorted list.

```

public void mergesort(Comparator<? super E> comp) {
    if (getSize() > 1) {
        head.setNext( mergesortImpl(comp, getSize(), head.next));
        version.increment();
    }
}

```

//invalidate active trackers for iteration

The recursive *mergesortImpl* method takes *comp*, the comparator to use to order the elements, *n*, the number of elements in this portion of this list, and *ptr*, the pointer to the first item in the list. It returns a reference to the first list item in the sorted list.

```
protected ListItem<E> mergesortImpl(Comparator<? super E> comp,
                                     int n, ListItem<E> ptr) {
    ListItem<E> ptrLeft = ptr;           //ptr is reference to first element on left half
    int numLeft = n/2;                     //number of elements in the left half
    for (int i = 0; i < numLeft-1; i++)    //move to middle of the list
        ptr = ptr.next;                  //ptr stops at last element on left half
    ListItem<E> ptrRight = ptr.next;      //ptrRight references first element on right
    ptr.next = getTail();                 //break into two lists
    if (numLeft > 1)                       //recursively sort left half
        ptrLeft = mergesortImpl(comp, numLeft, ptrLeft);
    if (n - numLeft > 1)                   //recursively sort right half
        ptrRight = mergesortImpl(comp, n - numLeft, ptrRight);
    return merge(comp, ptrLeft, ptrRight); //merge the two sorted lists into one
}
```

The *merge* method takes *comp*, the comparator to use to order the elements, *ptr1*, the pointer to the first item in the first list to merge, and *ptr2*, the pointer to the first item in the second list to merge. It merges the two lists and returns a reference to the first item in the merged list, which will be one of the two list items passed in as parameters.

```
protected ListItem<E> merge(Comparator<? super E> comp,
                             ListItem<E> ptr1, ListItem<E> ptr2) {
    ListItem<E> retVal = ptr1;           //reference head of merged list
    if (comp.compare((E) ptr1.data, (E) ptr2.data) ≤ 0) { //when smallest item in first list
        ptr1 = ptr1.next;                //move to second item (if any) from first list
    } else {                               //when smallest item in second list
        retVal = ptr2;                   //front of second list will be head of merged lists
        ptr2 = ptr2.next;                //move to second item (if any) from second list
    }
    ListItem<E> ptr = retVal;             //ptr references last item placed in merged list
    while (ptr1 != getTail() && ptr2 != getTail()) {
        if (comp.compare((E) ptr1.data, (E) ptr2.data) ≤ 0) {
            if (ptr.next != ptr1)         //when ptr at item from second list
                ptr.setNext(ptr1);        //adjust next pointer
            ptr1 = ptr1.next;             //move to next item from first list
        } else {                           //next element from second list is next smallest
            if (ptr.next != ptr2)         //when ptr at item from first list
                ptr.setNext(ptr2);        //adjust next pointer
            ptr2 = ptr2.next;             //move to next element from second list
        }
        ptr = ptr.next;                   //move forward in merged list
    }
    if (ptr2 != getTail())                 //if items remain in second list
        ptr.setNext(ptr2);                 //append to end of partially merged list
    else {                                 //still items in first list
        ptr.setNext(ptr1);                 //append to end of partially merged list
    }
```



```

        while (ptr.next != getTail())
            ptr = ptr.next;
        setLast(ptr);
    }
    return retVal;
}

```

//must also move ptr to last item
//of first list and hence of merged list
//reset last (currently end of second list)

15.5.3 Heap Sort

Using the same design technique as in Section 14.5, we use an anonymous class in the *getSorter* method that takes *comp*, the comparator to use for the elements. It returns a comparator defined over list items, that wraps the provided comparator.

```

private Comparator getSorter(final Comparator<? super E> comp){
    return new Comparator<ListItem>() {
        public int compare(ListItem a, ListItem b) {
            return comp.compare((E) a.data, (E) b.data);
        }
    };
}

```

To both preserve the trackers and avoid unnecessarily creating new list items when a list is rebuilt, we introduce the *resetList* method to update the list to be empty without changing the value of *size*. It must immediately be followed by the rebuilding of the list or *SIZE* will be violated. Another option, would be to reset *size* to 0 here, and have *addItemLast* increment *size*.

```

protected void resetList(){
    head.setNext(getTail());
    last = head;
}

```

//preserves EndsAtTail
//preserves Last

Recall that the *heapsort* method with no arguments uses the default comparator to order the elements.

```

public void heapsort() {
    heapsort(Objects.DEFAULT_COMPARATOR);
}

```

The *heapsort* method with an argument takes *comp*, the comparator to use to order the elements. This method uses *getSorter* to wrap *comp* for a comparator over list items.

```

public void heapsort(Comparator<? super E> comp) {
    heapsortImpl(getSorter(comp));
}

```

The *heapsortImpl* method that takes *sorter*, the comparator defined over list items to use, is the implementation of heap sort. The basic approach is as described in Section 11.4.3. The two main differences are that the list items are placed into the heap, and that the list (versus an array) must be rebuilt as the list items are extracted from the heap. As discussed on page 206, the time complexity of heap sort would be slightly reduced if the linear time method used by the BinaryHeap *addAll* method was used to build the heap instead of the successive insertions used here.

```

public void heapsortImpl(Comparator sorter) {
    BinaryHeap<Object> heap = new BinaryHeap<Object>(getSize(),
                                                new ReverseComparator(sorter));
    ListItem<E> ptr = head.next;                                //place all list items into the heap
    while (ptr != getTail()) {
        heap.add(ptr);
        ptr = ptr.next;
    }
    resetList();                                                //rebuild the list
    while (!heap.isEmpty())
        addItemLast((ListItem<E>) heap.extractMax());
    version.increment();                                        //invalidate active trackers for iteration
}

```

15.5.4 Tree Sort

Recall that the *treesort* method with no arguments uses the default comparator to order the elements.

```

public void treesort() {
    treesort(Objects.DEFAULT_COMPARATOR);
}

```

The *treesort* method with an argument takes *comp*, the comparator to use to order the elements. Since an inorder traversal, which is a recursive method, is the most efficient mechanism to iterate through a red-black tree in sorted order, a visitor is used to rebuild the list from the red-black tree.

```

public void treesort(Comparator<? super E> comp) {
    treesortImpl(getSorter(comp));
}

```

The *treesortImpl* method that takes *sorter*, the comparator to use is the implementation of tree sort.

```

protected void treesortImpl(Comparator sorter) {
    RedBlackTree<ListItem<E>> tree = new RedBlackTree<ListItem<E>>(sorter);
    ListItem<E> ptr = head.next;                                //insert all list items into a tree
    while (ptr != getTail()) {
        tree.add(ptr);
        ptr = ptr.next;
    }
    resetList();                                                //rebuild list using a visitor
    ptr = head;
    tree.accept(new Visitor<Object>() {
        public void visit(Object o) throws Exception {
            addItemLast((ListItem<E>) o);
        }
    });
}

```

15.5.5 Quicksort

This section assumes familiarity with the standard array-based implementation of quicksort as described in Section 11.4.5. For large collections that use a list-based representation, our approach avoids the space overhead of allocating an auxiliary array. Also, unlike the array-based quicksort, the list-based quicksort presented here is stable. Therefore, it would support two successive sorts using different comparators, with the final collection being primarily sorted by the second comparator, and secondarily sorted (to break ties), by the first comparator. The drawback is that the list-based implementation is slightly less time efficient due to the fact that one cannot access an arbitrary position in the list in constant time. If desired, one could use an approach like the one illustrated for tracked array (Chapter 14), where the list items are placed into an array that is sorted using a wrapped comparator, could be used.

For ease of exposition, we use the following notation throughout this subsection. For a list item reference, *loc*, we use the notation *loc* + 1 to refer to the list item that follows *loc*. Similarly, *loc* - 1 refers to the list element that follows *loc*. We say that list item *a* is less than list item *b* when *a.data* < *b.data*. We use the term **subcollection** to refer to the portion of the collection currently under consideration. This implementation of quicksort includes an optimization to sort only subcollections larger than a desired cutoff. In a final pass, insertion sort is used to complete the sort. Specifically, quicksort is only applied to a subcollection with a size greater than the constant *CUTOFF*.

final protected int CUTOFF = 7;

The *partition* method always will partition around the last element in the subcollection. If desired, randomization or the median-of-three technique could be used to select the partition element. Since the partition method has linear time complexity, introducing either of these techniques would not change the asymptotic time complexity. We have chosen not to illustrate either here since there is really no difference in the array-based versus list-based implementation of them.

The *partition* method used within the list-based implementation of quicksort must return (1) a reference to the element that precedes the last element on the left half, (2) the reference to the element before the (possibly different) last element in the subcollection, and (3) the number of elements on the left half of the partition. Because Java does not support call-by-reference, the *Divider* class is created to encapsulate these three variables for communication out of *partition*.

```
class Divider {
    ListItem<E> beforeEndOfLeft;           //reference to item preceding last in left half
    ListItem<E> beforeEnd;                 //reference to item preceding last in the subcollection
    int numLeft;                            //number of elements on the left half
}
```

The *quicksort* method with no arguments uses the default comparator.

```
public void quicksort() {
    quicksort(Objects.DEFAULT_COMPARATOR);
}
```

The *quicksort* method with on argument takes *comp*, the comparator to use to order the elements in the positional collection. It sorts the array in place.

```
public void quicksort(Comparator<? super E> comp) {
    if (getSize() > CUTOFF) {
        Divider mid = new Divider();           //just create one divider to reuse
        quicksortImpl(head, getPtrForPrevElement(getLast()), mid, getSize(), comp);
    }
}
```

```

    insertionsort();
}

```

Correctness Highlights: When *quicksortImpl* completes, all elements are within *CUTOFF* positions from their final position. The rest of the correctness argument follows from that of *insertionsort*, which is guaranteed to run in linear time since all the elements are within a constant number of positions from their final location. ■

The *quicksortImpl* method takes *beforeStart*, a reference to the list item just before the first one in the subcollection to sort, *beforeEnd*, a reference to the list item just before the last one in the subcollection to sort, *divider*, which holds information needed to divide the subcollection after partitioning, *size*, the number of elements in that subcollection, and *comp*, the comparator that defines the desired order of the elements. This method sorts the subcollection from *beforeStart* + 1 to *end* + 1 (inclusive).

Our implementation uses references before the start and end of the subcollection so that there is no need to move backwards within the list (except to initially reach the second to last element in the collection). This enables the implementation to be efficient for a singly linked list.

```

void quicksortImpl(ListItem<E> beforeStart, ListItem<E> beforeEnd, Divider divider,
    int size, Comparator<? super E> comp) {
    if (size > CUTOFF) { //insertion sort will be used to finish the sort
        partition(beforeStart, beforeEnd, divider, comp);
        int nLeft = divider.numLeft;
        beforeEnd = divider.beforeEnd;
        ListItem<E> beforeEndOfLeft = divider.beforeEndOfLeft;
        ListItem<E> mid = beforeStart.next; //value when nLeft = 0
        if (divider.numLeft > 1) //left side has >1 element
            mid = beforeEndOfLeft.next.next; //value when nLeft > 0
        quicksortImpl(beforeStart, beforeEndOfLeft, divider, nLeft, comp);
        quicksortImpl(mid, beforeEnd, divider, size - nLeft - 1, comp);
    }
}

```

Correctness Highlights: By the correctness of *partition*, all elements in the subcollection in the left half are < *mid.data*, and all elements in the subcollection in the right half are ≥ *mid.data*. Thus upon completion all elements are with *CUTOFF* positions of their final value. ■

The Hoare partition algorithm that is used for the *Array* class cannot be used here since it requires the ability to move backwards within the list in constant time per step. Instead we use the partition algorithm introduced by Nicolo Lomuto [23]. We first describe this partition method in terms of a standard array-based implementation that partitions *a[left], . . . , a[right]* where *swap(i,j)* swaps *a[i]* and *a[j]*. It returns the index for the final location of the pivot element. Figure 15.4 illustrates the execution of one call to *partition* on an array.

```

int lomutoPartition(int left, int right, Comparator<? super E> comp){
    E pivot = a[right]; //pivot around the right element
    int i = left - 1; //marks end of portion <= pivot
    for (int j = left; j ≤ right-1; j++){ //j marks start of portion to process
        if (comp.compare(a[j], pivot) ≤ 0) //when a[j] <= pivot
            swap(++i, j);
    }
}

```

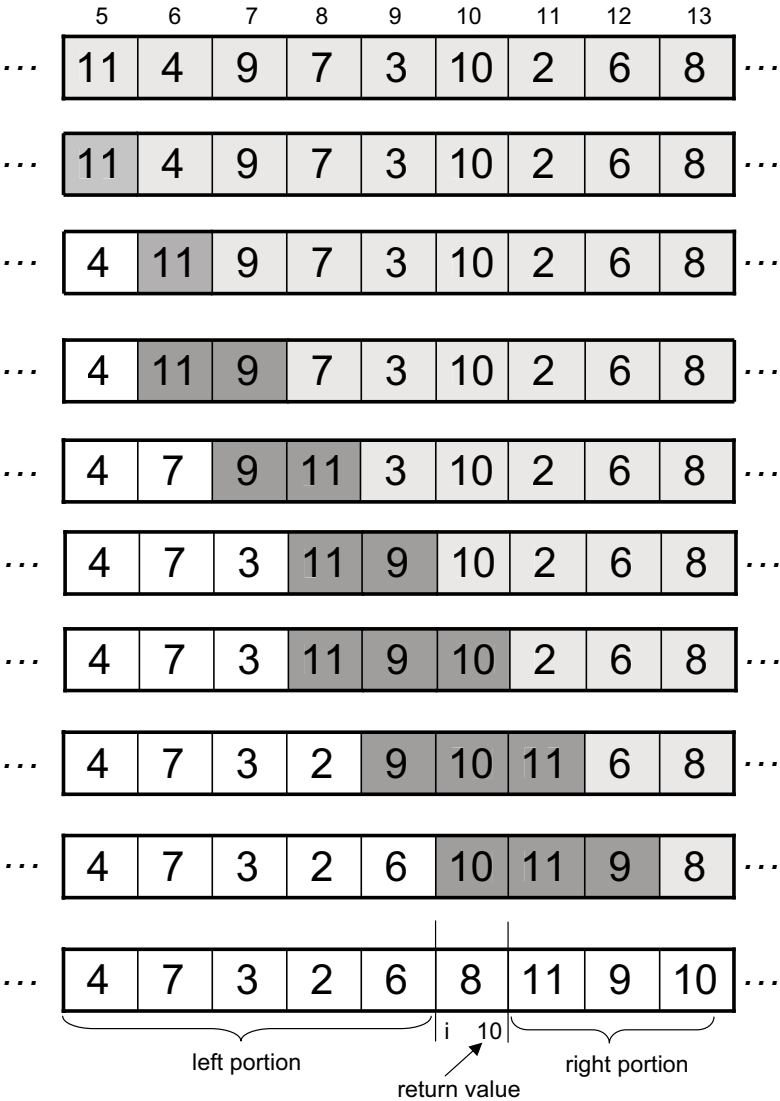
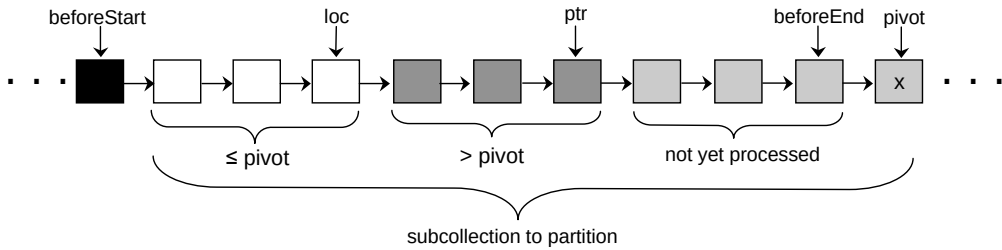


Figure 15.4 Sample execution of *partition* when *left* is 5 and *right* is 13. The array is shown at the start of each iteration of the for loop. The unshaded portion goes from the left to position i . The darkly shaded portion goes from position $i + 1$ to position $j - 1$. Finally, the lightly shaded portion is from position j to *right*. Note that after the left portion and right portion of the final subcollection are recursively sorted, the subcollection will be sorted.

**Figure 15.5**

This figure illustrates the invariant maintained by the *partition* method.

```

    }
    swap(++i, right);
    return i;
}

```

//swap pivot leftmost element > pivot
//return final position with pivot

In the standard array-based implementation of the Lomuto partition, as illustrated above, it is necessary to swap elements, rather simply inserting them at the correct position. This is because inserting into an array would require shifting the other elements in the collection to the right to create a gap for the inserted element, which would result in a quadratic time complexity. This swapping makes the array-based quicksort unstable. In contrast, the following list-based implementation is stable because elements are not swapped, but instead elements strictly less than the pivot element are simply inserted at the end of the left half of the subcollection, as they are encountered, keeping the sorting algorithm stable.

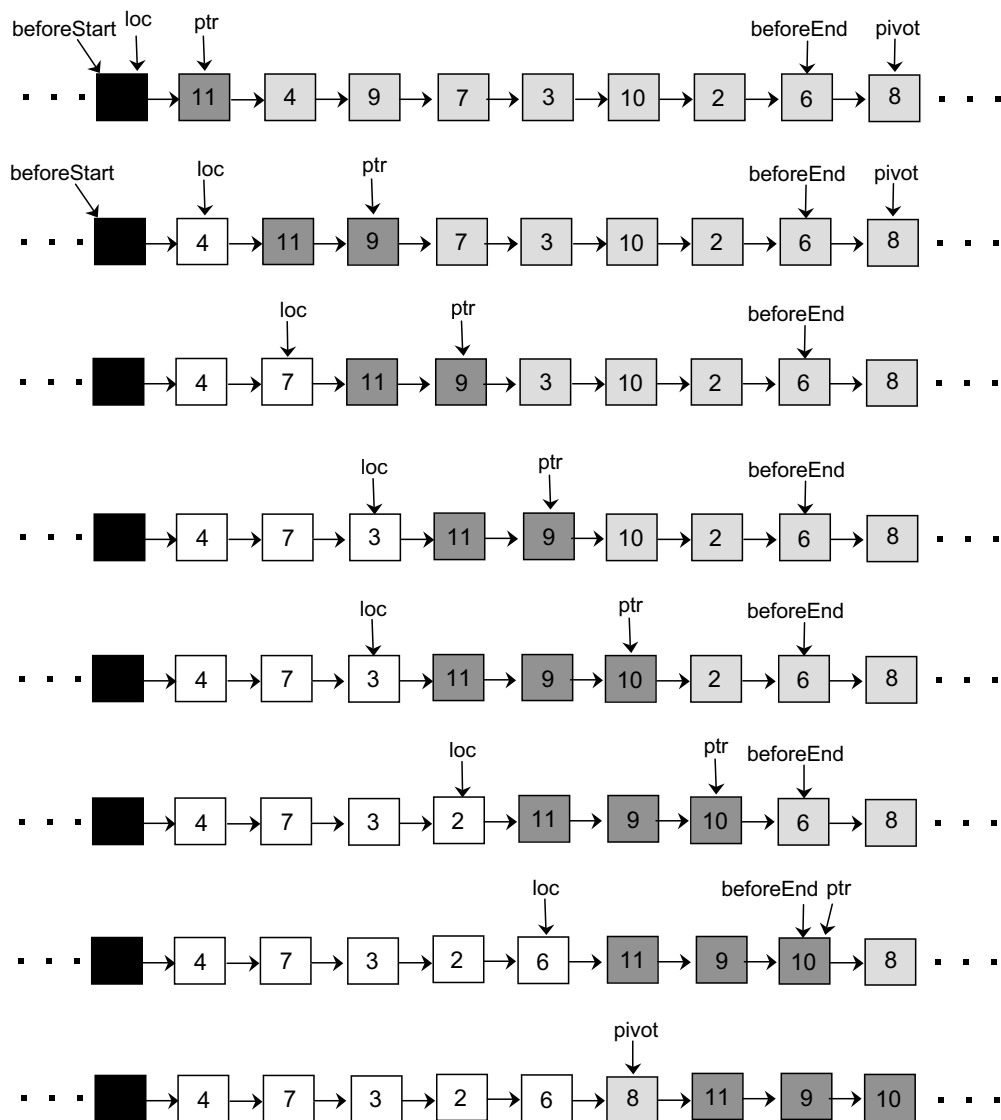
The *partition* method takes *loc*, a reference to the list item that references the start of the subcollection to partition, and *beforeEnd*, a reference to the second to last list item in the subcollection to partition. This method partitions the subcollection from *loc* + 1 to *beforeEnd* + 1 (inclusive). It modifies *loc* to reference the last element of the left half of the partition, and updates *divider* to reflect the new partition.

```

protected void partition(ListItem<E> loc, ListItem<E> beforeEnd, Divider divider,
    Comparator<? super E> comp) {
    E pivot = (E) beforeEnd.next.data;
    divider.numLeft = 0;
    ListItem<E> ptr = loc;
    while (ptr != beforeEnd) {
        if (comp.compare((E) ptr.next.data, pivot) ≤ 0) {
            if (ptr == loc) {
                divider.beforeEnd = ptr;
                ptr = ptr.next;
            } else {
                if (ptr.next == beforeEnd)
                    beforeEnd = ptr;
                if (loc == divider.beforeEnd & ptr.moveNextAfter(loc))
                    divider.beforeEnd = loc.next;
            }
            divider.numLeft++;
            divider.beforeEndOfLeft = loc;
            loc = loc.next;
        }
    }
}

```

//partition around end
//count number put on left half
//<= pivot
//move to the next element
//moving beforeEnd, so now
//ptr's value is the new beforeEnd
//put one more on left half
//always just before loc
//advance loc

**Figure 15.6**

Sample execution of our list-based *partition* method. Observe that the relative order of the elements before, and after, the pivot is unchanged.

```

    }
    else {
        divider.beforeEnd = ptr;
        ptr = ptr.next;           //move to the next element
    }
}
if (ptr == divider.beforeEnd)
    divider.beforeEnd = beforeEnd.next;
ListItem<E> newLast = beforeEnd;
if (getLast() == beforeEnd.next & beforeEnd.moveNextAfter(loc))
    setLast(newLast);
}

```

Correctness Highlights: Figure 15.5 illustrates the invariants that are maintained by the partition method. In particular, at the top of the for loop it always holds that:

- all items from *beforeStart* + 1 to *loc* have elements less than or equal to the pivot,
- all items from *loc* + 1 to *ptr* have elements are greater than the pivot, and
- the items from *ptr* + 1 to *beforeEnd* have not yet been compared to pivot.

Initially these hold vacuously since all elements in the subcollection have positions between *ptr* and *beforeEnd* + 1. It is easily verified that each execution of the while loop maintains it. Thus upon termination of the for loop, all items in positions *beforeStart* + 1 to *loc* hold elements that are $\leq$ *pivot*, and all items in positions *loc* + 1 to *beforeEnd* hold elements that are $>$ *pivot*. As illustrated in Figure 15.6, upon completion of the loop, all items in positions *beforeStart* + 1 to *loc* hold elements that are $\leq$ *pivot*, the pivot is in position *loc* + 1, and all items in positions *loc* + 2 to *beforeEnd* + 1 are $\geq$ *pivot*.

Thus *divider.beforeEnd* references the second to last element in the subcollection. It is also easily verified that *divider.numLeft* is the number of elements in the subcollection left of the pivot, and *divider.beforeEndOfLeft* references the item that is the second to last one of the portion of the subcollection left of the pivot. Finally, to preserve LAST, if the last element in the collection is being moved earlier in the array, then *last* is reset to the element preceding the one moved, which will be the last element in the collection when the method terminates. ■

15.5.6 Radix Sort

The fundamental difference between the array-based radix sort implementation in Section 11.4.6, and the list-based implementation presented here is the way in which the separation of elements by digit value is performed. Let *d* be the digit that is currently being processed, and let *L_v* be a list (bucket) of items that hold elements with value *v* for digit *d*. As the items are processed, those holding elements of value *v* are inserted to the front of *L_v*. Then, to ensure counting sort is both efficient and stable, the elements are inserted to the front of the singly linked list from *L_{b-1}*, *L_{b-2}*, ..., *L₁*, *L₀*. If *b* is the base of the digitizer, the only additional storage besides some local variables are the *b* head pointers for the buckets.

The *placeListItemFirst* takes *x*, a reference to the list item to insert to the front of the list. It is used to rebuild the list after the elements have been moved to the bucket according to the current digit, as the digits are considered from the least significant to most significant.


```

protected void placeListItemFirst(ListItem<E> x) {
    x.setNext(head.next);
    head.setNext(x);
    if (getLast() == head) //preserve Last
        setLast(x);
}

```

The *radixsort* method that takes *digitizer*, the digitizer to use in radix sort. It uses the algorithm described in Section 11.4.6, with the modification described above for keeping the bucket for each possible digit value as a list, and then inserting the contents of the buckets to the front of the resulting list in decreasing order of digit values.

```

public void radixsort(Digitizer<? super E> digitizer) {
    int numDigits = 0; //maximum number of digits in any element
    int b = digitizer.getBase();
    ListItem<E>[] headPtr = (ListItem<E>[]) new ListItem[b];
    for (E x : this) { //determine maximum number of digits
        int digits = digitizer.numDigits(x);
        if (digits > numDigits) //update numDigits
            numDigits = digits;
    }
    for (int d = 0; d < numDigits; d++) { //sort according to each digit d
        Arrays.fill(headPtr, null);
        while (head.next != getTail()) {
            int v = digitizer.getDigit((E) head.next.data, d);
            ListItem<E> temp = head.next.next;
            head.next.next = headPtr[v];
            headPtr[v] = head.next;
            head.next = temp;
        }
        for (int i = b-1; i ≥ 0; i--) { //rebuild the sorted list
            while (headPtr[i] != null) {
                ListItem<E> temp = headPtr[i].next;
                placeListItemFirst(headPtr[i]);
                headPtr[i] = temp;
            }
        }
    }
}

```

Correctness Highlights: It is easily seen that for each value of d in the for loop, the elements in the singly linked list on which this method is called will be sorted based on digit d , and that the sort is stable. The rest of the correctness is like that for the array-based implementation. ■

One possible optimization is to keep a reference to the last element in each bucket, and then the buckets can be spliced together to create the final list. In implementing this optimization it is important to insert the elements at the end of the bucket to keep the sort stable. However, this is easily done by using the reference to the end of each bucket.

15.5.7 Bucket Sort

In this section we describe our list-based implementation of bucket sort which uses the same basic approach as with radix sort. More specifically, a list is created for each bucket, the elements are inserted to the front of the appropriate list, and then the original list is rebuilt. As with the array-based implementation of Section 11.4.7, insertion sort is used to sort the elements within the buckets. The *bucketSort* method with an argument takes *bucketizer*, the bucketizer to use.

```

public void bucketSort(Bucketizer<? super E> bucketizer) {
    int b = bucketizer.getNumBuckets();
    ListItem<E>[] headPtr = (ListItem<E>[]) new ListItem[b];
    Arrays.fill(headPtr, null);
    while (head.next != getTail()) {
        int index = bucketizer.getBucket((E) head.next.data);
        ListItem<E> temp = head.next.next;
        head.next.next = headPtr[index];
        headPtr[index] = head.next;
        head.next = temp;
    }
    for (int i = b-1; i ≥ 0; i--) {                                //put back into list
        while (headPtr[i] != null) {
            ListItem<E> temp = headPtr[i].next;
            placeListItemFirst(headPtr[i]);
            headPtr[i] = temp;
        }
    }
    insertionSort();                                              //sort within buckets
}

```

Correctness Highlights: Like that for the array-based implementation of bucket sort. ■

15.6 Selection and Median Finding

In this chapter we present an algorithm to find the *i*th smallest element in a positional collection. Finding the median element is a special case of selection where $i = \lfloor (n - 1)/2 \rfloor$. We obtain an algorithm to find the median. As in Section 11.5 for arrays, the algorithm relies on the partition method used in quicksort. Again, our list-based implementation does not use randomization or the median-of-three method to select the pivot, but this could be done if desired.

The *repositionElementByRank* method that takes a single argument of *r*, the rank of the desired element in the sorted order of the elements. It has the side-effect of placing the element at position *r* in the collection. It returns the element at rank *r* when using the default comparator. It throws a *PositionOutOfBoundsException* when *r* is not a valid position.

```

public E repositionElementByRank(int r) {
    return repositionElementByRank(r, Objects.DEFAULT_COMPARATOR);
}

```

The *repositionElementByRank* method that takes *r*, the rank of the desired element in the sorted order of the elements, and *comp*, the comparator to use for determining the order, returns the element at rank *r* when using the given comparator. It has the side-effect of placing the element at position *r* in the collection. It throws a *NoSuchElementException* when the collection is empty or *r* is not a valid position.

```
public E repositionElementByRank(int r, Comparator<? super E> comp) {
    if (isEmpty() || r < 0 || r ≥ getSize())
        throw new NoSuchElementException();
    Divider mid = new Divider(); //just create one divider to reuse
    E result = repositionElementByRankImpl(head,
        getPtrForPrevElement(getLast()), mid, getSize(), r, comp);
    return result;
}
```

The *repositionElementByRankImpl* method takes six parameters: *beforeStart*, the list item just before the start of the partition, *beforeEnd*, the list item just before the last element of the partition, *div*, the Divider instance to be used in partition, *n*, the size of the partition, *r*, the rank of the desired element in the sorted order, and *comp*, the comparator used to order the elements. It returns the element at rank *r*, and requires $0 \leq r \leq n - 1$.

```
E repositionElementByRankImpl(ListItem<E> beforeStart, ListItem<E> beforeEnd,
    Divider div, int n, int r, Comparator<? super E> comp) {
    if (n == 1)
        return (E) beforeStart.next.data; //correct when n=1
    partition(beforeStart, beforeEnd, div, comp); //repositions endOfLeftHalf
    int nLeft = div.numLeft;
    ListItem<E> beforeEndOfLeft = div.beforeEndOfLeft;
    beforeEnd = div.beforeEnd;
    ListItem<E> mid = beforeStart.next; //value when nLeft = 0
    if (nLeft > 0) //left side has ≥ 1 element
        mid = beforeEndOfLeft.next.next; //value when nLeft > 0
    if (r == nLeft) {
        return (E) mid.data;
    } else if (r < nLeft) { //recurse on left half
        return repositionElementByRankImpl(beforeStart, beforeEndOfLeft,
            div, nLeft, r, comp);
    } else { //recurse on right half
        return repositionElementByRankImpl(mid, beforeEnd,
            div, n = nLeft - 1, r - nLeft - 1, comp);
    }
}
```

Correctness Highlights: Like that for the array-based implementation of Section 11.5. The collection is partitioned as in quicksort, but recursive calls are made only on one side of the pivot element, depending on whether the element of rank *r* is in the left or right portion. The remainder of the code just keeps track of the boundaries and middle of the subcollection. ■

15.7 Tracker Inner Class

`AbstractCollection<E>.AbstractLocator<E>` implements *Locator<E>*

↳ **Tracker** implements *PositionalCollectionLocator<E>*

The Tracker inner class for singly linked list implements the *PositionalCollectionLocator*. As discussed in more depth in Sections 2.5 and 5.8.3, a tracker will track a particular object and is robust to mutations made to the collection, including those made by other trackers or through public methods. Even if the elements are reordered (e.g., by applying a sort), the trackers will still track the same elements. Also, unless a critical mutator is executed, even when a tracked element is removed from the collection, the tracker still can be used to iterate through the other collection, or if desired can be used to begin tracking a different element. If iteration is attempted after a critical mutator is executed, a *ConcurrentModificationException* is thrown.

The *Tracker* class has a single instance variable, *ptr* that holds the reference to the list item containing the element being tracked. Observe that *head* serves the role of FORE since *head.next* references the list item holding the element at position 0.

```
ListItem<E> ptr; //pointer to tracked element
```

The constructor takes a single argument *startLoc*, a reference to the list item holding the element to track.

```
protected Tracker(ListItem<E> startLoc){
    ptr = startLoc;
}
```

Recall that *inCollection* returns *true* if and only if the tracked element is currently in the collection.

```
public boolean inCollection(){
    return ptr != head && ptr != getTail() && ptr.data != REMOVED;
}
```

| *Correctness Highlights:* Follows from PLACEMENT and REMOVED. ■

The *get* method returns the tracked element. It throws a *NoSuchElementException* when the tracker is not at an element in the collection.

```
public E get() {
    if (!inCollection())
        throw new NoSuchElementException();
    return (E) ptr.data;
}
```

| *Correctness Highlights:* Follows from PLACEMENT and the correctness of *inCollection*. ■

The tracker provides a simple mutator called *set* that takes *value*, the element to put at the current tracker location. It returns the element that had been stored at the tracker location and throws a *NoSuchElementException* when tracker is not at an element in the collection.

```
public E set(E value) {
    if (!inCollection())
        throw new NoSuchElementException();
}
```

```

    Object oldData = ptr.data;
    ptr.data = value;
    return (E) oldData;
}

```

Correctness Highlights: By the correctness of *inCollection*, an exception is appropriately thrown. By *PLACEMENT*, the collection element is updated. Finally, it is easily seen that this method preserves *PLACEMENT*. ■

We also provide an accessor *getCurrentPosition* that returns the position of the tracker within the collection. It throws a *NoSuchElementException* when the tracker is not at an element in the collection. This method takes constant time for an element not in the collection. For an element in the collection, this method takes time proportional to the position of the tracked element, which in the worst-case is linear time.

```

public int getCurrentPosition() {
    if (!inCollection())
        throw new NoSuchElementException();
    int i = 0;
    for (ListItem<E> loc = head.next; loc != ptr; loc = loc.next)
        i++;
    return i;
}

```

Correctness Highlights: By the correctness of *inCollection*, the exception is appropriately thrown. The rest of the correctness follows from *PLACEMENT*. ■

The internal method *skipRemovedElements* performs a similar task as it does for the tracked array (see Chapter 14). To help illustrate the need for this method, the top of Figure 15.7 illustrates the scenario that occurs if a sequence of consecutive elements are removed from the positional collection illustrated in Figure 15.1. To advance the tracker, we follow the *next* references until a non-deleted list item is reached. For the sake of efficiency, path compression is performed. Namely, if a sequence of *next* pointers is followed for a sequence of unreachable list items, then once a reachable ListItem x_p is found, all of the *next* pointers followed are updated to refer to x_p . Figure 15.7 shows a singly linked list before and after executing *skipRemovedElements(loc)*.

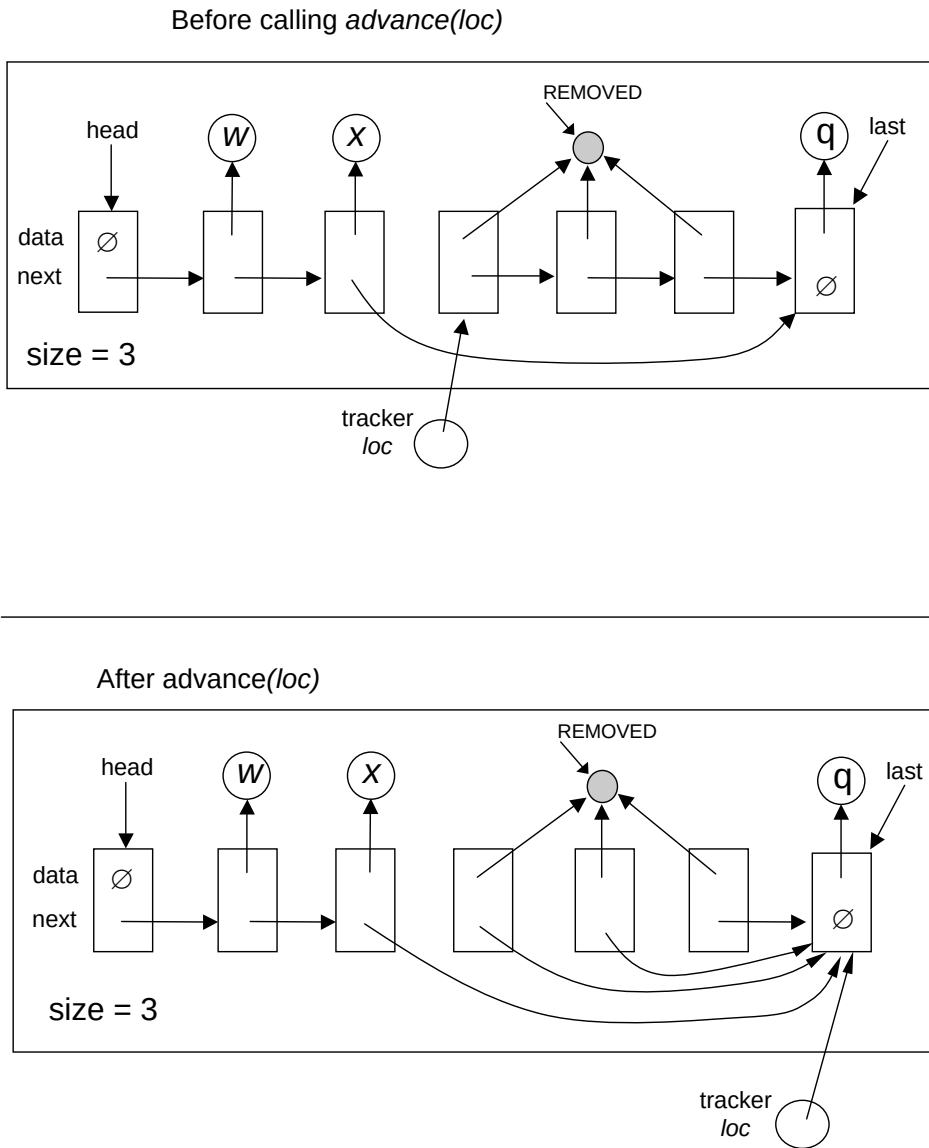
The *skipRemovedElements* method takes *ptr*, a pointer to a list item, and returns a pointer to the first reachable list item, possibly *ptr* itself if it is reachable. More formally, $x_{pos(ptr)}$ is returned.

```

protected ListItem<E> skipRemovedElements(ListItem<E> ptr) {
    if (ptr == getTail() || ptr.data != REMOVED)
        return ptr;
    ptr.next = skipRemovedElements(ptr.next);
    return ptr.next;
}

```

Correctness Highlights: We prove this using induction on the number of recursive calls made by *skipRemovedElements*. If *ptr.data* = *REMOVED* or *ptr* is at the tail then the correct value is returned, and since no updates occur, all properties are preserved. We now consider the inductive step. For a list item *ptr* that is not in use, by definition $pos(ptr) = pos(ptr.next)$. Thus the correct

**Figure 15.7**

Top: A singly linked list with three removed elements, one of which is tracked. Bottom: The same collection after executing *loc.advance()*.

value is returned, and the update to *ptr.next* preserves *REMOVED*.

In addition, every unreachable list item *y* accessed so *y.next* references the list node for the reachable element $x_{pos(y)}$. ■

Recall the *advance* method moves the tracker to the next location and returns *true* if and only if the tracker moves to a valid position. It throws an *AtBoundaryException* when the tracker is already at AFT since there is no place to advance.

```
public boolean advance() throws ConcurrentModificationException {
    if (ptr == getTail())
        throw new AtBoundaryException("Already after end.");
    checkValidity();
    if (ptr.data == REMOVED)
        ptr = skipRemovedElements(ptr);
    else
        ptr = ptr.next;
    return ptr != getTail();
}
```

Correctness Highlights: If the tracker is currently at *tail* then the required exception is thrown. We first consider the case when the tracked element is not in the collection (which by *PLACEMENT* and *REMOVED* occurs when *ptr.data* = *REMOVED*). Suppose the tracker is between positions *p* − 1 and *p*. By *REMOVED* and the correctness of *skipRemovedElements*, *ptr* is updated to track the element at position *p*, which is the correct behavior for *advance*.

Next, we consider the case when the tracked element is in the collection. In this case, *ptr* is advanced to the next element in the collection. In both cases, *true* is correctly returned as long as the updated tracker location is not AFT which occurs exactly when *ptr* = *tail*. ■

The *retreat* method updates the tracker to the previous element in the collection. It throws an *AtBoundaryException* when the tracker is at *FORE*. It returns *true* if and only if the tracker is at an element in the collection after retreating.

```
public boolean retreat() throws ConcurrentModificationException {
    if (ptr == head)
        throw new AtBoundaryException("Already before start.");
    checkValidity();
    if (ptr != getTail() && ptr.data == REMOVED)
        ptr = skipRemovedElements(ptr);
    ptr = getPtrForPrevElement(ptr);
    return (ptr != head);
}
```

Correctness Highlights: If the tracker is currently at *head*, then the required exception is thrown. We first consider when *skipRemovedElements* is called. By *PLACEMENT* and *REMOVED*, this situation occurs exactly when the tracked item had been removed from the collection at some point (since *ptr* is not at the head or tail). Suppose the tracker is between positions *p* − 1 and *p*. By *REMOVED* and the correctness of *skipRemovedElements*, *ptr* is updated to track the element at position *p*.

We now consider when the *getPtrForPrevElement* is executed. At this point, the tracked element is in the collection, but it needs to move back one position. By the correctness of *getPtrForPrevElement*, the tracker is updated to the correct position. In both cases, *true* is correctly returned as long as the updated tracker is not *FORE*, which occurs exactly when *ptr = head*. ■

The *hasNext* method returns *true* if there is some element after the current tracker position.

```
public boolean hasNext() throws ConcurrentModificationException {
    checkValidity();
    return ptr != getTail() && ptr.next != getTail() &&
        skipRemovedElements(ptr) != getTail();
}
```

Correctness Highlights: Clearly if *ptr* or *ptr.next* references the tail, then there are no elements beyond the tracker, and *false* should be returned. The remainder of the correctness follows from that of *skipRemovedElements*, which returns the next element in the iteration order, or the tail if there are no more elements. ■

The *addAfter* method takes *value*, the element to add. It adds the new element after the tracked element, and returns a tracker to the new element.

```
public PositionalCollectionLocator<E> addAfter(E value)
    throws ConcurrentModificationException {
    if (ptr.data == REMOVED)
        throw new NoSuchElementException();
    return insertAfter(ptr, value);
}
```

Correctness Highlights: If the tracked element had been removed, then an exception is appropriately thrown. Otherwise, the correctness follows from the *insertAfter* method. ■

The *remove* method that removes the tracked element from the collection. Since there are no previous pointers, this method takes $O(p)$ time, where p is the position of the tracked element. It throws a *NoSuchElementException* when the tracked element has already been removed.

```
public void remove() throws ConcurrentModificationException {
    if (!inCollection())
        throw new NoSuchElementException();
    removeNext(getPtrForPrevElement(ptr));
}
```

Correctness Highlights: By the correctness of *inCollection*, the exception is appropriately thrown. The rest follows from the correctness of *removeNext* method and *getPtrForPrevElement*. ■

A problem with the *remove* method is that to remove the tracked item from the singly linked list structure, it is necessary to back up the previous element by search forward from head, which can take linear time. Therefore, additional methods are provided specifically for a singly linked list tracker to improve performance of removing items during iteration. Because there is no previous pointer, these methods allow the tracker to look one element ahead in order to possibly remove the next item while the tracker still retains the pointer to the list item that precedes the one to be removed.

The *getNextElement* method returns the element after the tracked element. It throws a *NoSuchElementException* when the tracked element is not in the collection, or if current element is at position $size - 1$.

```
public E getNextElement() throws ConcurrentModificationException {
    if (ptr.data == REMOVED || !hasNext())
        throw new NoSuchElementException();
    return (E) ptr.next.data;
}
```

Correctness Highlights: By *REMOVED* and the correctness of *hasNext*, the *NoSuchElementException* is appropriately thrown. The rest of the correctness follows from that of *PLACEMENT*. ■

The *removeNextElement* method removes the element that follows the tracked element. It throws a *NoSuchElementException* when the tracked element is not in the collection, or if current element is at position $size - 1$.

```
public void removeNextElement() throws ConcurrentModificationException {
    if (ptr.data == REMOVED || !hasNext())
        throw new NoSuchElementException();
    removeNext(ptr);
}
```

Correctness Highlights: By *REMOVED* and the correctness of *hasNext*, the *NoSuchElementException* is appropriately thrown. The rest of the correctness follows from that of *removeNext*. ■

15.8 Performance Analysis

The asymptotic time complexities of all public methods for the singly linked list data structure, as compared to that of the circular array, are given in [Table 15.8](#). The asymptotic time complexities for all the public methods of the locator class are given in [Table 15.9](#).

Clearly the constructors take constant time as well as the *iterator* and *iteratorAtEnd* method. The *ensureCapacity* and *trimToSize* do nothing, so they both take constant time.

The *addFirst* method inserts the new element to the front of the list which can be done in constant time by adjusting two references. Since *last* references the last list item, *addLast* and *add* take constant time. Similarly, *removeFirst* takes constant time.

The *getPtr* and *getPtrForPrevElement* methods take $O(p + 1)$ time when accessing the object in the collection at position p since the list must be traversed starting at the head reference to reach a given element (even if its position is known) and exactly $p + 1$ next references are examined. Thus the public methods *get*, *set*, *add(p,o)*, and *remove(p)* have asymptotic time complexity of $O(p + 1)$. The *swap(p1,p2)* method takes $O(\max(p1, p2))$ time with the dominant cost being the time to reach position $p1$ or $p2$, whichever comes later. The remainder of the *swap* method takes constant time.

Similarly, *removeRange(i,j)* takes $O(i)$ time to reach position i , and then $O(j - i + 1)$ time complexity to remove the elements. Hence, the overall asymptotic time complexity for *removeRange(i,j)* is $O(j + 1)$. Although *addLast* takes constant time, *removeLast* takes linear time since it is necessary to locate the element in position $n - 1$ so that its *next* reference can be updated.

method	singly linked list time complexity	circular array time complexity
constructors	$O(1)$	$O(1)$
<i>add(o)</i>	$O(1)$	$O(1)$
<i>addFirst(o)</i>	$O(1)$	$O(1)$
<i>addLast(o)</i>	$O(1)$	$O(1)$
<i>ensureCapacity(x)</i>	$O(1)$	$O(x)$
<i>removeFirst()</i>	$O(1)$	$O(1)$
<i>trimToSize()</i>	$O(1)$	$O(n)$
<i>add(p,o)</i>	$O(p + 1)$	$O(\min(p + 1, n - p))$
<i>get(p)</i>	$O(p + 1)$	$O(1)$
<i>remove(p)</i>	$O(p + 1)$	$O(\min(p + 1, n - p))$
<i>removeRange(p,q)</i>	$O(q + 1)$	$O(\min(q + 1, n - p))$
<i>set(p,o)</i>	$O(p + 1)$	$O(1)$
<i>swap(p1,p2)</i>	$O(\max(p1, p2))$	$O(1)$
<i>accept(v)</i>	$O(n)$	$O(n)$
<i>addAll(c)</i>	$O(n)$	$O(n)$
<i>bucketSort()</i>	$O(n)$ expected	$O(n)$ expected
<i>clear()</i>	$O(n)$	$O(n)$
<i>contains(o)</i>	$O(n)$	$O(n)$
<i>getLocator(o)</i>	$O(n)$	$O(n)$
<i>positionOf(o)</i>	$O(n)$	$O(n)$
<i>remove(o)</i>	$O(n)$	$O(n)$
<i>removeLast()</i>	$O(n)$	$O(1)$
<i>repositionElementByRank(i)</i>	$O(n)$ expected	$O(n)$ expected
<i>toString()</i>	$O(n)$	$O(n)$
<i>radixSort()</i>	$O(d(n + b))$	$O(d(n + b))$
<i>retainAll(c)</i>	$O(n c)$	$O(n(c + n))$
<i>heapSort()</i>	$O(n \log n)$	$O(n \log n)$
<i>mergeSort()</i>	$O(n \log n)$	$O(n \log n)$
<i>quickSort()</i>	$O(n \log n)$ expected	$O(n \log n)$ expected
<i>treeSort()</i>	$O(n \log n)$	$O(n \log n)$
<i>insertionSort()</i>	$O(n^2)$	$O(n^2)$

Table 15.8 Summary of the time complexities when using a singly linked list to implement the PositionalCollection ADT as compared to using a circular array.

locator method	singly linked list complexity	circular array complexity
<i>addAfter(o)</i>	$O(1)$	$O(n)$
<i>advance()</i>	$O(1)$	$O(1)$
<i>get()</i>	$O(1)$	$O(1)$
<i>getNextElement()</i>	$O(1)$	$O(1)$
<i>hasNext()</i>	$O(1)$	$O(1)$
<i>next()</i>	$O(1)$	$O(1)$
<i>removeNextElement()</i>	$O(1)$	$O(n)$
<i>set(o)</i>	$O(1)$	$O(1)$
<i>getCurrentPosition()</i>	$O(n)$	$O(1)$
<i>remove()</i>	$O(n)$	$O(n)$
<i>retreat</i>	$O(n)$	$O(1)$

Table 15.9 Summary of the time complexities for the Locator methods when using a singly linked list to implement the PositionalCollection ADT as compared to using a circular array.

In the worst case, *contains* and *positionOf* take linear time since they must iterate through the list until the desired element or position is reached. Also *accept*, *addAll*, *clear*, *getLocator*, *toString*, and *traverseForVisitor* methods take linear time since they iterate through all elements spending constant time at each element. The time complexity for the sorting algorithms is the same as for the implementations in the Array class (see Chapter 11).

The *addAll* method takes linear time since it takes constant time to add each element. Finally, the *retainAll* method iterates through the collection removing elements that do not occur in the provided collection. For each element of *c* it takes, in the worst case, $O(|c|)$ time for the call to *contains* on collection *c*, and constant time to remove the element if it is not in *c*. Thus, the overall time complexity of *retainAll* is $O(n \cdot |c|)$ time.

We now analyze the methods in the Tracker class. Methods *get*, *inCollection*, and *set* take constant time since they only require access to the list item that is referenced by the tracker. The time required for *skipRemovedElements* is proportional to the number of removed elements that are accessed before reaching an element in the collection. In the worst-case it could have linear cost. However, since this method modifies all references followed to now reference an element in the collection, the cost of this method is constant when amortized over *remove* calls made for the collection.

When the tracker is at an element in the collection, *addAfter*, *advance*, *getNextElement*, *hasNext*, *next*, and *removeNextElement* take constant time since they can use the *next* reference to locate the element that follows. Thus their time is dominated by *skipRemovedElements*. So while these methods have worst-case linear time, their amortized cost is constant. Also, the worst-case cost would only be high if a long sequence of adjacent elements were removed. Similarly, *addAfter* takes linear time. In contrast, observe that with an array-based implementation *addAfter* and *removeNextElement* take linear time in the worst-case.

Since the only way to locate an element that precedes another in the collection is to move forward from the front of the list, the *retreat* method takes worst-case linear time. Similarly, to remove an element from the collection one must locate the element before it so that its *next* pointer can be updated. While *last* enables the last element to be reached in constant time, the *removeLast* method takes linear time since it requires updating the *next* reference in the second to last list item. Like *retreat*, *remove*, and *getCurrentPosition* take $O(p)$ time if the tracker is at position *p* since the list must be traversed from the head to reach the previous element, or to count the number of preceding elements. Thus, these methods have worst-case linear cost.

15.9 Quick Method Reference

SinglyLinkedList Public Methods

- p. 221 *SinglyLinkedList()*
- p. 98 void **accept**(Visitor<? super E> v)
- p. 227 void **add**(E value)
- p. 227 void **add**(int p, E value)
- p. 100 void **addAll**(Collection<? extends E> c)
- p. 122 void **addFirst**(E value)
- p. 122 void **addLast**(E value)
- p. 227 Locator<E> **addTracked**(E value)
- p. 244 void **bucketSort**(Bucketizer<? super E> bucketizer)
- p. 100 void **clear**()
- p. 97 boolean **contains**(E value)
- p. 99 void **ensureCapacity**(int capacity)
- p. 223 E **get**(int p)
- p. 96 int **getCapacity**()
- p. 96 Comparator<? super E> **getComparator**()
- p. 98 E **getEquivalentElement** (E target)
- p. 231 PositionalCollectionLocator<E> **getLocator**(E element)
- p. 96 int **getSize**()
- p. 235 void **heapSort**()
- p. 235 void **heapSort**(Comparator<? super E> comp)
- p. 235 void **heapSortImpl**(Comparator sorter)
- p. 232 void **insertionSort**()
- p. 233 void **insertionSort**(Comparator<? super E> comp)
- p. 96 boolean **isEmpty**()
- p. 231 PositionalCollectionLocator<E> **iterator**()
- p. 231 PositionalCollectionLocator<E> **iteratorAt**(int pos)
- p. 231 PositionalCollectionLocator<E> **iteratorAtEnd**()
- p. 233 void **mergesort**()
- p. 233 void **mergesort**(Comparator<? super E> comp)
- p. 224 int **positionOf**(E value)
- p. 237 void **quicksort**()
- p. 237 void **quicksort**(Comparator<? super E> comp)
- p. 243 void **radixSort**(Digitizer<? super E> digitizer)
- p. 230 boolean **remove**(E value)
- p. 229 E **remove**(int p)
- p. 230 E **removeFirst**()
- p. 230 E **removeLast**()
- p. 229 void **removeRange**(int fromPos, int toPos)
- p. 244 E **repositionElementByRank**(int r)
- p. 245 E **repositionElementByRank**(int r, Comparator<? super E> comp)
- p. 100 void **retainAll**(Collection<E> c)
- p. 224 E **set**(int p, E value)
- p. 226 void **swap**(int a, int b)
- p. 97 Object[] **toArray**()
- p. 97 E[] **toArray**(E[] array)
- p. 98 String **toString**()
- p. 236 void **treeSort**()
- p. 236 void **treeSort**(Comparator<? super E> comp)
- p. 99 void **trimToSize**()

SinglyLinkedList Internal Methods

- p. 232 void **addItemLast**(ListItem<E> x)
- p. 97 int **compare**(E e1, E e2)
- p. 97 boolean **equivalent**(E e1, E e2)
- p. 222 ListItem<E> **getLast**()
- p. 223 ListItem<E> **getPtr**(int p)
- p. 223 ListItem<E> **getPtrForPrevElement**(E value)
- p. 222 ListItem<E> **getPtrForPrevElement**(ListItem<E> ptr)
- p. 235 Comparator **getSorter**(Comparator<? super E> comp)
- p. 222 ListItem<E> **getTail**()
- p. 222 void **initialize**()
- p. 226 PositionalCollectionLocator<E> **insertAfter**(ListItem<E> ptr, E value)
- p. 234 ListItem<E> **merge**(Comparator<? super E> comp, ListItem<E> ptr1, ListItem<E> ptr2)
- p. 234 ListItem<E> **mergesortImpl**(Comparator<? super E> comp, int n, ListItem<E> ptr)
- p. 221 ListItem<E> **newListItem**(E value)
- p. 240 void **partition**(ListItem<E> loc, ListItem<E> beforeEnd, Divider divider, Comparator<? super E> comp)
- p. 242 void **placeListItemFirst**(ListItem<E> x)
- p. 238 void **quicksortImpl**(ListItem<E> beforeStart, ListItem<E> beforeEnd, Divider divider, int size, Comparator<? super E> comp)
- p. 228 E **removeNext**(ListItem<E> ptr)
- p. 245 E **repositionElementByRankImpl**(ListItem<E> beforeStart, ListItem<E> beforeEnd, Divider div, int n, int r, Comparator<? super E> comp)
- p. 235 void **resetList**()
- p. 224 void **setLast**(ListItem<E> last)
- p. 225 void **swapAfter**(ListItem<E> aPrev, ListItem<E> bPrev)
- p. 99 void **traverseForVisitor**(Visitor<? super E> v)
- p. 236 void **treesortImpl**(Comparator sorter)
- p. 98 void **writeElements**(StringBuilder s)

SinglyLinkedList.Tracker Public Methods

- p. 250 PositionalCollectionLocator<E> **addAfter**(E value)
- p. 249 boolean **advance**()
- p. 246 E **get**()
- p. 247 int **getCurrentPosition**()
- p. 251 E **getNextElement**()
- p. 250 boolean **hasNext**()
- p. 101 void **ignoreConcurrentModifications**(boolean ignore)
- p. 101 void **ignorePriorConcurrentModifications**()
- p. 246 boolean **inCollection**()
- p. 101 E **next**()
- p. 250 void **remove**()
- p. 251 void **removeNextElement**()
- p. 249 boolean **retreat**()
- p. 246 E **set**(E value)

SinglyLinkedList.Tracker Internal Methods

- p. 246 *Tracker*(ListItem<E> startLoc)
- p. 101 void **checkValidity**()
- p. 247 ListItem<E> **skipRemovedElements**(ListItem<E> ptr)
- p. 101 void **updateVersion**()

Chapter 16

Doubly Linked List Data Structure

package `collection.positional`

`AbstractCollection<E>` implements *Collection<E>*

└ `SinglyLinkedList<E>` implements *PositionalCollection<E>*, *Tracked<E>*

└ `DoublyLinkedList<E>` implements *PositionalCollection<E>*, *Tracked<E>*

Uses: Java references

Used By: `Buffer` (Chapter 17), `KDTree` (Chapter 47), `QuadTree` (Chapter 48), `AdjacencyMatrixRepresentation` (Chapter 54), `AdjacencyListRepresentation` (Chapter 55)

Strengths: The doubly linked list is the only positional collection data structure that provides amortized constant time methods for all of the *PositionalCollectionLocator* methods except *getCurrentPosition*. Furthermore, it is a tracked implementation. Since a doubly linked list maintains a pointer from each element to the one that precedes it, an element that is known to be located near the end of the collection can be efficiently located by traversing backwards from the tail.

Weaknesses: The space requirement is roughly $3n$ pointers for a collection holding n elements. Also it takes linear time to locate an element via its position when the element is near the middle of the collection.

Critical Mutators: *heapsort*, *insertionsort*, *mergesort*, *quicksort*, *radixsort*, *repositionElementByRank*, *swap*, *treesort*

Competing Data Structures: The singly linked list (Chapter 15) can reduce the space usage without any cost in the time complexity if the application does not ever access elements (via position) that are near the end of the collection, add or remove elements near the end of the collection, use the locator *retreat* method, or use the locator *remove* method.

An array-based implementation is a better choice if it is important to be able to efficiently access elements via position, and if it is acceptable for the add and remove methods to take linear time for elements toward the middle of the collection. An array-based data structure also enables *getCurrentPosition* to be implemented in constant time. Finally, an array makes the most efficient use of space, especially if the size of the collection is known in advance.

16.1 Internal Representation

The doubly linked list is an extension of singly linked list that adds a previous pointer to allow constant time access to the previous list item.

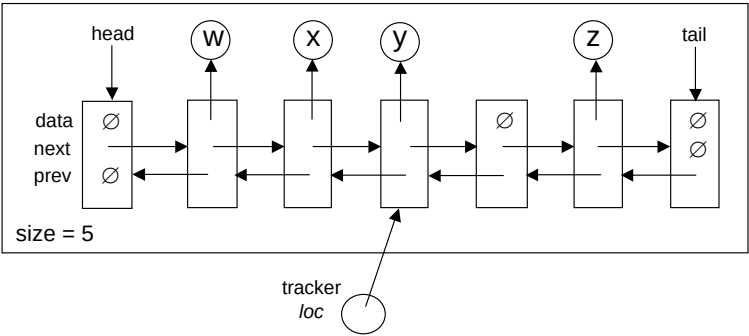


Figure 16.1
A populated example of a doubly linked list that holds the positional collection $\langle w, x, y, \text{null}, z \rangle$.

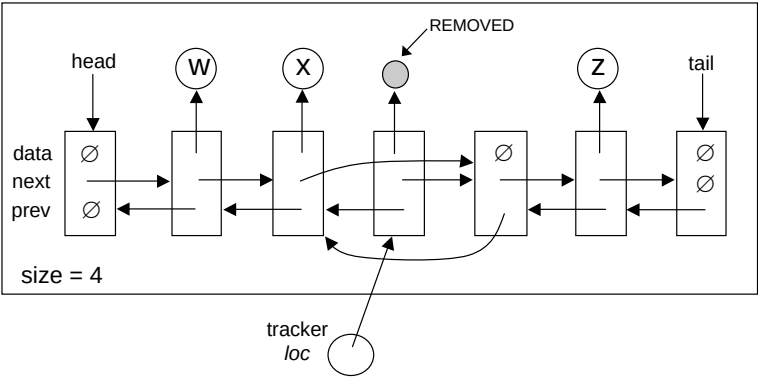


Figure 16.2
A populated example of the doubly linked list from Figure 16.1 after the element referenced by the locator is removed.

Instance Variables and Constants: In addition to the inherited *size* and *head* variables, the `DoublyLinkedList` adds a *tail* sentinel to avoid special cases when adding or removing elements at the end of the collection. The tail sentinel serves the role of AFT.

```
DLListItem<E> tail; //tail sentinel
```

Populated Example: Figure 16.1 shows the internal representation for a doubly linked list for the positional collection $\langle w, x, y, \text{null}, z \rangle$ ($n=5$). A tracker is shown referring to the element, *y*, in position 2. Figure 16.2 shows the internal representation that results when *y* is removed.

Abstraction Function: The abstraction function for doubly linked list is the same as that for a singly linked list.

16.2 Representation Properties

We preserve the representation properties of the singly linked list, and add one more property.

PREVIOUSPOINTER: Let x be a reference to a list item for the element in position p in the collection. The pointer $x.prev$ references the list item for the element in position $p - 1$, where *head* is at position -1 , and *tail* is at position *size*. In other words, $x.prev.next = x$.

16.3 Doubly Linked List Item Inner Class

ListItem<E>

└─ **DLLItem**<E>

We define the *DLLItem* class that extends *ListItem* by adding an instance variable, *prev*, that is a reference to the previous element in the list. The *setNext* method is overridden to preserve PREVIOUSPOINTER.

```
DLLItem<E> prev;

DLLItem(Object data) {
    super(data);
}

protected void setNext(ListItem<E> next) {
    this.next = next;
    ((DLLItem<E>) next).prev = this;
}
```

Correctness Highlights: This method preserves PREVIOUSPOINTER. ■

16.4 Doubly Linked List Methods

The most significant change from the singly linked list is that the previous reference can be used any time there is a need to iterate backwards in the list. The *DLLItem setNext* method which takes care of adjusting the *previous* pointer. We also override the constructor and the factory method that creates a new list item.

16.4.1 Constructors and Factory Methods

The *newListItem* method takes *value*, the desired element, and returns a new *DLLItem* holding the given element.

```
protected DLLItem<E> newListItem(E value) {
    return new DLLItem<E>(value);
}
```

The *initialize* method updates this collection to be empty.


```

void initialize(){
    tail = newListItem(null);
    super.initialize();
}

```

Correctness Highlights: SIZE is satisfied since $size = n = 0$. The collection holds no elements, so PLACEMENT vacuously holds. Since *head* is initialized to *getTail()*, ENDSATTAIL holds. Finally, there are no unreachable list items created by this method, so REMOVED vacuously holds. LAST also holds since it should reference the *head* sentinel when the collection is empty. ■

16.4.2 Representation Accessors

Recall that for a singly linked list, the value of *tail* was considered to be *null*. Here, we override the *getTail* to return the tail sentinel.

```

protected ListItem<E> getTail() {
    return tail;
}

```

The *getLast* method returns a reference to the last item in the list. In a doubly linked list, the previous pointer of the tail sentinel replaces the use of the instance variable *last*.

```

protected ListItem<E> getLast() {
    return tail.prev;
}

```

16.4.3 Algorithmic Accessors

We introduce an internal method *findFromBack* that takes *p*, the position of the element to find. It returns a reference to the *DLLListItem* for the element in the collection at position *p*. It requires that *p* is a valid position in the collection. This method enables a more efficient search to be performed whenever $p \geq size/2$.

```

private DLLListItem<E> findFromBack(int p){
    DLLListItem<E> current = tail.prev;
    for (int loc = size-1; loc > p; loc--)
        current = current.prev;
    return current;
}

```

Correctness Highlights: By the requirement that $0 \leq p \leq size-1$, this method will terminate. The correctness follows from PREVIOUSPOINTER. ■

Recall that the *getPtr* method takes *p*, a valid user position or -1, and returns a reference to the element in the collection at position *p* or the head sentinel if *p* is -1. We override this method to make use of the previous pointer to improve the time complexity.

```

protected DLListItem<E> getPtr(int p){
    if (p ≥ size/2)
        return findFromBack(p);
    else
        return (DLListItem<E>) super.getPtr(p);
}

```

Correctness Highlights: Follows from that of the *findFromBack* method and that of the *SinglyLinkedList getPtr(int p)* method that is used when $p < \text{size}/2$. ■

Recall, the internal *getPrevPtr* method where *ptr*, reference to an element in the collection returns a reference to the list item just before that referenced by *ptr* (possibly the *head* sentinel). It requires that *ptr* is a pointer to an element of the collection. By using the previous pointer this method runs in constant time versus the worst-case linear time performance for a singly linked list.

```

protected ListItem<E> getPtrForPrevItem(ListItem<E> ptr){
    return ((DLListItem<E>) ptr).prev;
}

```

Correctness Highlights: Follows from PREVIOUSPOINTER. ■

16.4.4 Representation Mutators

The *setLast* method takes *ptr*, a reference to the list item that is now the last one in the list. It sets the previous references of the tail sentinel to the given value.

```

protected void setLast(ListItem<E> ptr) {
    tail.prev = (DLListItem<E>) ptr;
}

```

16.5 Performance Analysis

[Table 16.3](#) compares the asymptotic time complexities of all public methods for a doubly linked list to that of a singly linked list and a circular array. The asymptotic time complexities for all of the methods of the locator class are given in [Table 16.4](#).

We only discuss the methods whose time complexity changes from *SinglyLinkedList*. The *getPrevPtr(ptr)* method is implemented in constant time. Furthermore, the *getPtr(p)* method takes $O(\min p + 1, n - p)$ time by starting at the tail sentinel and moving backwards if $p > n/2$. This improves the time complexity of the methods *get*, *set*, *add(p,v)*, and *remove(p)* to $O(\min(p+1, n-p))$.

The *swap(p1,p2)* method takes $O(\min(p1 + 1, n - p1) + \min(p2 + 1, n - p2))$ since positions *p1* and *p2* can both be located from either the front or back of the list. The remainder of the swap method takes constant time.

Similarly, *removeRange(i,j)* has time complexity of $O(\min(i, n - j - 1))$ to reach either element *i* or *j*, followed by $O(j - i + 1)$ time to remove the elements. Hence, the overall asymptotic time complexity for *removeRange(i,j)* is $O(\min(j + 1, n - i))$.

Since it takes constant time to locate the element that precedes any element in the collection, *retreat* and *remove* take constant time.

method	DLL time complexity	SLL time complexity	circular array time complexity
constructor, capacity x	$O(1)$	$O(1)$	$O(x)$
<i>ensureCapacity(x)</i>	$O(1)$	$O(1)$	$O(x)$
<i>trimToSize()</i>	$O(1)$	$O(1)$	$O(n)$
<i>add(o)</i>	$O(1)$	$O(1)$	$O(1)$
<i>addFirst(o)</i>	$O(1)$	$O(1)$	$O(1)$
<i>addLast(o)</i>	$O(1)$	$O(n)$	$O(1)$
<i>removeFirst()</i>	$O(1)$	$O(1)$	$O(1)$
<i>removeLast()</i>	$O(1)$	$O(n)$	$O(1)$
<i>add(p,v)</i>	$O(d(p))$	$O(p+1)$	$O(d(p)+1)$
<i>get(p)</i>	$O(d(p))$	$O(p+1)$	$O(1)$
<i>remove(p)</i>	$O(d(p))$	$O(p+1)$	$O(d(p)+1)$
<i>set(p,o)</i>	$O(d(p))$	$O(p+1)$	$O(1)$
<i>swap(p,q)</i>	$O(d(p)+d(q))$	$O(q+1)$	$O(1)$
<i>removeRange(p,q)</i>	$O(d(p)+d(q)+q-p+1)$	$O(q)$	$O(\min(q+1, p-i))$
<i>accept(v)</i>	$O(n)$	$O(n)$	$O(n)$
<i>addAll(c)</i>	$O(n)$	$O(n)$	$O(n)$
<i>bucketSort()</i>	$O(n)$ expected	$O(n)$ expected	$O(n)$ expected
<i>clear()</i>	$O(n)$	$O(n)$	$O(n)$
<i>contains(o)</i>	$O(n)$	$O(n)$	$O(n)$
<i>getElementAtRank(i)</i>	$O(n)$ expected	$O(n)$ expected	$O(n)$ expected
<i>getLocator(o)</i>	$O(n)$	$O(n)$	$O(n)$
<i>positionOf(o)</i>	$O(n)$	$O(n)$	$O(n)$
<i>remove(o)</i>	$O(n)$	$O(n)$	$O(n)$
<i>toString()</i>	$O(n)$	$O(n)$	$O(n)$
<i>radixSort()</i>	$O(d(n+b))$	$O(d(n+b))$	$O(d(n+b))$
<i>heapSort()</i>	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
<i>mergeSort()</i>	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
<i>quickSort()</i>	$O(n \log n)$ expected	$O(n \log n)$ expected	$O(n \log n)$ expected
<i>treeSort()</i>	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
<i>insertionSort()</i>	$O(n^2)$	$O(n^2)$	$O(n^2)$
<i>retainAll(c)</i>	$O(n c)$	$O(n c)$	$O(n(c +n))$

Table 16.3 Summary of the time complexities when using a doubly linked list (DLL) as compared to using a singly linked list (SLL) or circular array. We let $d(p) = \min(p+1, n-p)$, which is the minimum number of references that must be accessed to reach position p from either end of the list. For the circular array, $d(p)$ corresponds to the number of elements that must be shifted to insert or remove an element from position p .

locator method	DLL time complexity	SLL time complexity	circular array time complexity
<i>addAfter(o)</i>	$O(1)$	$O(1)$	$O(n)$
<i>advance()</i>	$O(1)$	$O(1)$	$O(1)$
<i>get()</i>	$O(1)$	$O(1)$	$O(1)$
<i>getNextElement()</i>	$O(1)$	$O(1)$	$O(1)$
<i>hasNext()</i>	$O(1)$	$O(1)$	$O(1)$
<i>next()</i>	$O(1)$	$O(1)$	$O(1)$
<i>remove()</i>	$O(1)$	$O(n)$	$O(n)$
<i>removeAfter()</i>	$O(1)$	$O(1)$	$O(n)$
<i>removeNext()</i>	$O(1)$	$O(1)$	$O(n)$
<i>retreat</i>	$O(1)$	$O(n)$	$O(1)$
<i>set(o)</i>	$O(1)$	$O(1)$	$O(1)$
<i>getCurrentPosition()</i>	$O(n)$	$O(n)$	$O(1)$

Table 16.4 Summary of the time complexities for the Locator methods when using an DoublyLinkedList (DLL) to implement the Positional Collection ADT as compared to using a circular array or a SinglyLinkedList (SLL).

16.6 Quick Method Reference

DoublyLinkedList Public Methods

- p. 98 void **accept**(Visitor<? super E> v)
- p. 227 void **add**(E value)
- p. 227 void **add**(int p, E value)
- p. 100 void **addAll**(Collection<? extends E> c)
- p. 122 void **addFirst**(E value)
- p. 122 void **addLast**(E value)
- p. 227 Locator<E> **addTracked**(E value)
- p. 244 void **bucketSort**(Bucketizer<? super E> bucketizer)
- p. 100 void **clear**()
- p. 97 boolean **contains**(E value)
- p. 99 void **ensureCapacity**(int capacity)
- p. 223 E **get**(int p)
- p. 96 int **getCapacity**()
- p. 96 Comparator<? super E> **getComparator**()
- p. 98 E **getEquivalentElement** (E target)
- p. 231 PositionalCollectionLocator<E> **getLocator**(E element)
- p. 96 int **getSize**()
- p. 235 void **heapsort**()
- p. 235 void **heapsort**(Comparator<? super E> comp)
- p. 235 void **heapsortImpl**(Comparator sorter)
- p. 232 void **insertionsort**()
- p. 233 void **insertionsort**(Comparator<? super E> comp)
- p. 96 boolean **isEmpty**()
- p. 231 PositionalCollectionLocator<E> **iterator**()
- p. 231 PositionalCollectionLocator<E> **iteratorAt**(int pos)
- p. 231 PositionalCollectionLocator<E> **iteratorAtEnd**()
- p. 233 void **mergesort**()
- p. 233 void **mergesort**(Comparator<? super E> comp)

- p. 224 int **positionOf**(E value)
- p. 237 void **quicksort**()
- p. 237 void **quicksort**(Comparator<? super E> comp)
- p. 243 void **radixsort**(Digitizer<? super E> digitizer)
- p. 230 boolean **remove**(E value)
- p. 229 E **remove**(int p)
- p. 230 E **removeFirst**()
- p. 230 E **removeLast**()
- p. 229 void **removeRange**(int fromPos, int toPos)
- p. 244 E **repositionElementByRank**(int r)
- p. 245 E **repositionElementByRank**(int r, Comparator<? super E> comp)
- p. 100 void **retainAll**(Collection<E> c)
- p. 224 E **set**(int p, E value)
- p. 226 void **swap**(int a, int b)
- p. 97 Object[] **toArray**()
- p. 97 E[] **toArray**(E[] array)
- p. 98 String **toString**()
- p. 236 void **treesort**()
- p. 236 void **treesort**(Comparator<? super E> comp)
- p. 99 void **trimToSize**()

DoublyLinkedList Internal Methods

- p. 232 void **addItemLast**(ListItem<E> x)
- p. 97 int **compare**(E e1, E e2)
- p. 97 boolean **equivalent**(E e1, E e2)
- p. 260** DLLItem<E> **findFromBack**(int p)
- p. 222 ListItem<E> **getLast**()
- p. 223 ListItem<E> **getPtr**(int p)
- p. 223 ListItem<E> **getPtrForPrevElement**(E value)
- p. 222 ListItem<E> **getPtrForPrevElement**(ListItem<E> ptr)
- p. 261** ListItem<E> **getPtrForPrevItem**(ListItem<E> ptr)
- p. 235 Comparator **getSorter**(Comparator<? super E> comp)
- p. 222 ListItem<E> **getTail**()
- p. 222 void **initialize**()
- p. 226 PositionalCollectionLocator<E> **insertAfter**(ListItem<E> ptr, E value)
- p. 234 ListItem<E> **merge**(Comparator<? super E> comp, ListItem<E> ptr1, ListItem<E> ptr2)
- p. 234 ListItem<E> **mergesortImpl**(Comparator<? super E> comp, int n, ListItem<E> ptr)
- p. 221 ListItem<E> **newListItem**(E value)
- p. 240 void **partition**(ListItem<E> loc, ListItem<E> beforeEnd, Divider divider, Comparator<? super E> comp)
- p. 242 void **placeListItemFirst**(ListItem<E> x)
- p. 238 void **quicksortImpl**(ListItem<E> beforeStart, ListItem<E> beforeEnd, Divider divider, int size, Comparator<? super E> comp)
- p. 228 E **removeNext**(ListItem<E> ptr)
- p. 245 E **repositionElementByRankImpl**(ListItem<E> beforeStart, ListItem<E> beforeEnd, Divider div, int n, int r, Comparator<? super E> comp)
- p. 235 void **resetList**()
- p. 224 void **setLast**(ListItem<E> last)
- p. 225 void **swapAfter**(ListItem<E> aPrev, ListItem<E> bPrev)
- p. 99 void **traverseForVisitor**(Visitor<? super E> v)
- p. 236 void **treesortImpl**(Comparator sorter)
- p. 98 void **writeElements**(StringBuilder s)

Chapter 17

Buffer ADT and Its Implementation

package collection.positional

Buffer<E>

This chapter studies the Buffer ADT which is a specialization of a positional collection. It is sometimes called a **double-ended queue** or **deque** (pronounced “deck”). In a buffer, items are added or removed only at the ends of the collection. Our buffer implementations wraps a positional collection, to which the actual work is delegated. We consider bounded and unbounded buffers as well as tracked and untracked versions of each.

Uses: CircularArray (Chapter 12), DynamicCircularArray (Section 13.6), DoublyLinkedList (Chapter 16)

Used By: Queue (Chapter 18)

Strengths: Provides a more efficient implementation of a positional collection when the natural interface for the application requires only adding and removing elements from the front and back of the collection.

Weaknesses: A buffer provides access only at the front and back of the collection.

Competing Data Structures: A stack (Chapter 19) is a better choice if it is desirable to enforce that all elements are added or removed from the front of the collection. This produces an abstraction of a last-in, first-out line. Likewise, a queue (Chapter 18) should be considered if an access pattern of a first-in, first-out line is appropriate for the application.

17.1 Internal Representation

Elements are stored in a positional collection *buffer*. An instance variable, *bound* holds the limit (if any) on the size of the buffer. Conceptually, an unbounded buffer has no limit on its size, and *getCapacity* returns infinity. In reality, it is limited by the amount of available memory.

```
static final int DEFAULT_CAPACITY = 8;           //default capacity
private PositionalCollection<E> buffer;         //stores the buffer elements
private int bound;                             //max number of elements to be held in the buffer
```

Abstraction Function: Let B be a buffer. The abstraction function

$$AF(B) = \langle u_0, u_1, \dots, u_{n-1} \rangle \text{ such that } u_p = a.get(p) \text{ and } n = a.getSize().$$

Design Notes: Our buffer implementation provides an example of the **adaptor design pattern** (Section C.2) to make an existing object adapt to a different interface, and the **builder design pattern** (Section C.4) to create a polymorphic implementation in which methods delegate work to a wrapped object that adheres to a general interface (in this case, a positional collection). The adaptor hides the functionality of the wrapped object that is not desirable for the wrapper type. Namely, while the buffer uses a positional collection, it hides methods that would allow access (except through a tracker) to elements in the middle of the buffer. To create the wrapped positional collection, the constructor includes logic to select an appropriate positional collection to support the desired buffer properties. The allocation of the object with *new* plays the builder role, and the constructor plays the director role.

Optimizations: Implementing the buffer as a wrapper adds some overhead. A very slight gain in efficiency (at the expense of abstraction) could be realized by having the application directly use a positional collection instead.

17.2 Representation Properties

For the buffer, queue, and stack, the properties are the same as for the positional collection that is wrapped. Since the correctness arguments all follow fairly directly from the wrapped class, we do not give any representation properties or correctness arguments for these classes.

17.3 Methods

We now present the internal and public methods for the Buffer class.

17.3.1 Constructors

The most general constructor takes three arguments, *capacity*, the initial capacity for the underlying positional collection, *bounded*, a boolean which is true if and only if the buffer should be bounded at the given capacity, and *tracked*, a boolean which is true if and only if the buffer should support a tracker. This constructor creates a buffer satisfying the specification of the given parameters. The wrapped positional collection data structure is selected as follows.

- **Tracked buffer:** For this situation, a doubly linked list is used for the positional collection data structure. The only other positional collection data structures that support a tracker are a singly linked list and a tracked array. The reason we have not selected a singly linked list is that it takes linear time to remove the last item (since there is no efficient way to locate the item at position *size-2* to update its next pointer). The advantage of the doubly linked list over a tracked array is that dynamic arrays incur space and time overhead when resizing is needed. The key advantage of a tracked array is the ability to locate the item at any position in constant time. Since the *Buffer* interface only requires locating the first or last element, there is not a need for this additional capability.
- **Untracked buffer that is bounded:** For this situation, the circular array data structure is ideal since it can minimize space usage. The size for the underlying array is defined by the bound given for the buffer.

- **Untracked buffer that is unbounded:** For this situation, a dynamic circular array is used since it does not incur the additional overhead required to support a tracker and enables efficient insertion and deletion at both ends of the buffer. The other natural alternative is a doubly linked list. We use a dynamic circular array since it is more efficient in its space usage, even though occasional resizing may be needed.

The instance variable, *bound*, is set to *java.lang.Integer.MAX_VALUE* when the buffer is unbounded, and *capacity* when the buffer is bounded. When an array-based representation is used, *capacity* is used as the initial capacity of the underlying array.

```
public Buffer(int capacity, boolean bounded, boolean tracked){
    if (bounded)
        bound = capacity;
    else
        bound = java.lang.Integer.MAX_VALUE;
    if (tracked)
        buffer = new DoublyLinkedList<E>();
    else if (bounded)
        buffer = new CircularArray<E>(capacity);
    else
        buffer = new DynamicCircularArray<E>(capacity);
}
```

The other constructors use default values for the unspecified parameters. The constructor with no arguments creates an unbounded, untracked buffer with a default initial capacity.

```
public Buffer(){
    this(DEFAULT_CAPACITY, false, false);
}
```

The constructor with a single argument, *capacity*, the initial capacity for the underlying positional collection, creates an unbounded, untracked buffer with the given initial capacity.

```
public Buffer(int capacity){
    this(capacity, false, false);
}
```

The constructor with two arguments, *capacity*, the initial capacity for the underlying positional collection, and *bounded*, a boolean which is true if and only if the buffer should be bounded, creates an untracked buffer with the specified parameters.

```
public Buffer(int capacity, boolean bounded){
    this(capacity, bounded, false);
}
```

17.3.2 Trivial Accessors

The only trivial accessors supported here are *isEmpty* and *getSize*. Others could be easily added. These methods just delegate to the corresponding method of the wrapped positional collection. Recall that *isEmpty* returns *true* if and only if no elements are stored in the collection.


```

public boolean isEmpty() {
    return buffer.isEmpty();
}

```

The method *getSize* returns the number of elements stored in the buffer.

```

public int getSize() {
    return buffer.getSize();
}

```

17.3.3 Algorithmic Accessors

The algorithmic accessors of the *PositionalCollection* interface supported here are *toString* and *contains*. As with the trivial accessors, they call the corresponding method on *a*.

```

public String toString() {
    return buffer.toString();
}

```

The method *contains* takes *value*, the target, and returns *true* if and only if an equivalent value exists in the buffer.

```

public boolean contains(E value) {
    return buffer.contains(value);
}

```

We replace the *PositionalCollection* *get* method with two methods specific to a buffer. The method *getFirst* returns the first element in the buffer, leaving the buffer unchanged.

```

public E getFirst() {
    return buffer.get(0);
}

```

Similarly, *getLast* returns the last element in the buffer, leaving the buffer unchanged.

```

public E getLast() {
    return buffer.get(buffer.getSize()-1);
}

```

Although other algorithmic accessors could be supported, the buffer is not intended to provide general access by position. If such methods are desired, then a data structure that implements the *PositionalCollection* interface should be used instead.

17.3.4 Content Mutators

The content mutators we support are *addFirst*, *addLast*, *removeFirst*, *removeLast*, and *clear*. For a bounded buffer, an exception is thrown if there is an attempt to add an element to a full buffer. The rest of the computation is performed by using the corresponding method of the wrapped positional collection.

The buffer method *addFirst* takes *element*, the element that is to be inserted at the front of the buffer. It throws an *AtCapacityException* when a bounded buffer is full.

```

public void addFirst(E element) {
    if (buffer.getSize() == bound)                //always false for unbounded buffer
        throw new AtCapacityException();
    buffer.addFirst(element);
}

```

The buffer method *addLast* takes *element*, the element that is to be inserted at the end of the buffer. It throws an *AtCapacityException* when a bounded buffer is full.

```

public void addLast(E element) {
    if (buffer.getSize() == bound)
        throw new AtCapacityException();
    buffer.addLast(element);
}

```

The *removeFirst* method removes the first element in the buffer, and returns the removed element. It throws a *NoSuchElementException* when the buffer is empty.

```

public E removeFirst() {
    return buffer.removeFirst();
}

```

Similarly, *removeLast* removes the last element in the buffer and returns the element that was removed. It throws a *NoSuchElementException* when the buffer is empty.

```

public E removeLast() {
    return buffer.removeLast();
}

```

The *clear* method removes all elements from the buffer.

```

public void clear() {
    buffer.clear();
}

```

17.3.5 Locator Initializers

The *iterator* method creates a new locator that starts just before the first item in the buffer. One might argue that providing an iterator for a buffer is a violation of the abstraction since all access to a buffer is supposed to be at the start and end of a collection. However, applications do occasionally need to examine the contents of a buffer, and may want to remove tracked elements. If the application requires that users of the iterator not be permitted to add and remove elements from the buffer, a non-mutating iterator could be returned instead.

```

public PositionalCollectionLocator<E> iterator() {
    return buffer.iterator();
}

```

17.4 Performance Analysis

The constructor takes constant time when a doubly linked list is wrapped, and time proportional to the capacity when an array-based representation is used. The asymptotic time complexity of the other methods is as follows:

method	time complexity
constructor	$O(1)$ or $O(n)$
<i>addFirst(o)</i>	$O(1)$
<i>addLast(o)</i>	$O(1)$
<i>getFirst()</i>	$O(1)$
<i>getLast()</i>	$O(1)$
<i>getSize()</i>	$O(1)$
<i>isEmpty()</i>	$O(1)$
<i>iterator()</i>	$O(1)$
<i>removeFirst()</i>	$O(1)$
<i>removeLast()</i>	$O(1)$
<i>clear()</i>	$O(n)$
<i>contains(o)</i>	$O(n)$
<i>toString()</i>	$O(n)$

Since we have not wrapped the locator, the time complexity for any locator method is the same as that for the wrapped class. Specifically, if an element is removed from the middle of the buffer through a locator, it will take constant time if a doubly linked list is the underlying representation, and linear time if either a circular array or dynamic circular array is used. If it is important for the application that there be constant time support for mutations through a locator, then a doubly linked list should be wrapped, even if the buffer need not be tracked.

17.5 Quick Method Reference

Buffer Public Methods

- p. 267 *Buffer()*
- p. 267 *Buffer(int capacity)*
- p. 267 *Buffer(int capacity, boolean bounded)*
- p. 267 *Buffer(int capacity, boolean bounded, boolean tracked)*
- p. 268 void **addFirst**(E element)
- p. 269 void **addLast**(E element)
- p. 269 void **clear**()
- p. 268 boolean **contains**(E value)
- p. 268 E **getFirst**()
- p. 268 E **getLast**()
- p. 268 int **getSize**()
- p. 267 boolean **isEmpty**()
- p. 269 PositionalCollectionLocator<E> **iterator**()
- p. 269 E **removeFirst**()
- p. 269 E **removeLast**()
- p. 268 String **toString**()

Chapter 18

Queue ADT and Implementation

package collection.positional

Queue<E>

Uses: Buffer (Chapter 17)

Used By: PairingHeap (Chapter 27), AbstractGraph (Chapter 53), AdjacencyMatrixRepresentation (Chapter 54), AbstractWeightedGraph (Chapter 57), case study on maintaining request quorums for Byzantine agreement (Section 5.8.1)

Strengths: A queue is a more specialized abstraction than a buffer, and is therefore more natural for some applications. By providing restricted access, it helps prevent accidental misuse of the data structure.

Weaknesses: Elements can only be inserted to the back and removed from the front. That is, it can only support a first-in-first-out (FIFO) line.

Competing Data Structures: If a last-in-first-out (LIFO) line is needed, then a stack (Chapter 19) should be considered. If more general access at the ends of a positional collection are needed then consider a buffer (Chapter 17).

18.1 Internal Representation

The queue is implemented using a buffer. We limit our discussion here to providing the documentation for the methods that are different than those supported for the buffer. All of the methods are directly delegated to the appropriate buffer method.

```
Buffer<E> buffer;
```

18.2 Methods

We now present the methods for the Queue class. The constructors have the same structure as the Buffer constructors.

```
public Queue(){this(Buffer.DEFAULT_CAPACITY, false, false);}
public Queue(int capacity){this(capacity, false, false);}
```

```

public Queue(int capacity, boolean bounded){this(capacity, bounded, false);}

public Queue(int capacity, boolean bounded, boolean tracked){
    buffer = new Buffer<E>(capacity, bounded, tracked);
}

```

The *clear*, *contains*, *getSize*, *isEmpty*, and *toString* methods directly delegate to the wrapped buffer.

```

public void clear() {buffer.clear();}

public boolean contains(E value) {return buffer.contains(value);}

public int getSize() {return buffer.getSize();}

public boolean isEmpty() {return buffer.isEmpty();}

public String toString() {return buffer.toString();}

```

As discussed for the buffer, exposing an iterator is in some sense a violation of the FIFO interaction with a queue. If necessary, a non-mutating iterator could be used to prevent an application from modifying the middle of the collection.

```

public Locator<E> iterator() {return buffer.iterator();}

```

The *peek* method returns the first object in the queue. The queue is not changed.

```

public E peek() {
    return buffer.getFirst();
}

```

The *enqueue* methods takes *element*, the new element to insert, and inserts *element* at the end of the queue. It throws an *AtCapacityException* when a bounded queue is full.

```

public void enqueue(E element) {
    buffer.addLast(element);
}

```

The *dequeue* method removes the element from the front of the queue. It returns the object that was removed, and throws a *NoSuchElementException* when the queue is empty. This method performs the equivalent computation as the buffer *addLast* method.

```

public E dequeue() {
    return buffer.removeFirst();
}

```

18.3 Performance Analysis

The constructor takes constant time when a doubly linked list is wrapped, and time proportional to the capacity when an array-based representation is used. The asymptotic time complexity of the other methods is as follows:

method	time complexity
constructor	$O(1)$ or $O(n)$
<i>dequeue()</i>	$O(1)$
<i>enqueue(o)</i>	$O(1)$
<i>getSize()</i>	$O(1)$
<i>isEmpty()</i>	$O(1)$
<i>iterator()</i>	$O(1)$
<i>peek()</i>	$O(1)$
<i>clear()</i>	$O(n)$
<i>contains(o)</i>	$O(n)$
<i>toString()</i>	$O(n)$

Since we have not wrapped the locator, the time complexity for any locator method is the same as that for the wrapped class. Specifically, if an element is removed from the middle of the queue through a locator it will take constant time if a list is the underlying representation, and linear time if either a circular array or dynamic circular array is used. If the application requires constant time support for mutations through a locator, then a list should be wrapped, even if the queue need not be tracked.

18.4 Quick Method Reference

Queue Public Methods

- p. 271 *Queue()*
- p. 271 *Queue(int capacity)*
- p. 271 *Queue(int capacity, boolean bounded)*
- p. 272 *Queue(int capacity, boolean bounded, boolean tracked)*
- p. 272 *void clear()*
- p. 272 *boolean contains(E value)*
- p. 272 *E dequeue()*
- p. 272 *void enqueue(E element)*
- p. 272 *int getSize()*
- p. 272 *boolean isEmpty()*
- p. 272 *Locator<E> iterator()*
- p. 272 *E peek()*
- p. 272 *String toString()*

Chapter 19

Stack ADT and Implementation

package collection.positional

Stack<*E*>

Uses: Array (Chapter 11), DynamicArray (Chapter 13), SinglyLinkedList (Chapter 15)

Used By: Object Pool (Section 5.5)

Strengths: More specialized than a buffer, a stack is natural for applications that insert and remove elements only at one end of the buffer. That is, a stack implements a last-in-first-out (LIFO) line. By providing restricted access, it helps prevent accidental misuse of the data structure. Also, since all updates to the stack are made at one end, an efficient array-based implementation can be achieved with a non-circular array, and an efficient list-based implementation can be achieved with a singly linked list.

Weaknesses: Elements can only be inserted and removed from one end of the buffer.

Competing Data Structures: If a first-in-first-out (FIFO) line is needed, then a queue (Chapter 18) should be considered. If more general access at the ends of a positional collection is needed, then consider a buffer (Chapter 17).

Stack

19.1 Internal Representation

Traditionally, one can view a stack like a physical stack of objects, where new objects are added and removed from the top. The bottom of the stack holds the object that was added farthest in the past. Like the queue, a stack could be implemented using a buffer. However, a more efficient implementation can be obtained by exploiting the fact that all inserts and removals can be made only from one end of the stack. In particular, for an untracked stack, a non-circular array is sufficient, where the top of the stack is at the back of the array. This approach enables all updates to be performed in constant time without the overhead associated with a circular array. For an untracked stack, a singly linked list is used where the top of the stack is at the front of the list, where efficient insertion and removal can be performed. Using a doubly linked list would waste space, since there would be no improvement in time complexity.

The instance variable, *stack*, is the wrapped positional collection. An instance variable, *bound*, holds the bound (if any) on the number of elements to be held in the stack.

```
private PositionalCollection<E> stack;  
private int bound;
```

```
//stores the stack elements  
//maximum number of elements
```

19.2 Methods

We now present the internal and public methods for the Stack class. Using the same approach as for the buffer (Chapter 17), the constructor chooses the underlying data structure. When an array-based implementation is used, it need not be circular. Also, since the singly linked list has little space overhead, we use it (as opposed to a tracked array) when a tracked implementation is needed.

The most general constructor takes three arguments, *capacity*, the initial capacity for the stack, *bounded*, a boolean that indicates if the stack should be bounded, and *tracked*, a boolean that indicates if a tracked implementation is needed. The constructor creates a stack satisfying the given requirements.

```
public Stack(int capacity, boolean bounded, boolean tracked){
    bound = bounded? capacity : java.lang.Integer.MAX_VALUE;
    if (tracked)                                //use singly linked list if tracked
        stack = new SinglyLinkedList<E>();
    else {                                       //use an array-based approach if not tracked
        if (bounded)                           //use an array if bounded and untracked
            stack = new Array<E>(capacity);
        else                                   //use a dynamic array if unbounded and untracked
            stack = new DynamicArray<E>(capacity);
        }
    }
```

The constructor with no arguments creates an bounded, untracked stack with a default initial capacity.

```
public Stack(){
    this(Buffer.DEFAULT_CAPACITY, true, false);
}
```

The constructor with a single argument, *capacity*, the initial capacity for the stack, creates an unbounded, untracked stack with the given capacity.

```
public Stack(int capacity){
    this(capacity, false, false);
}
```

The constructor with two arguments, *capacity*, the initial capacity for the stack, and *bounded*, which indicates if the stack should be bounded, creates an untracked stack satisfying the given requirements.

```
public Stack(int capacity, boolean bounded){
    this(capacity, bounded, false);
}
```

The *isEmpty*, *getSize*, *toString*, *contains*, *clear*, and *iterator* methods are delegated to the wrapped class.

```
public boolean contains(E value) {return stack.contains(value);}
public void clear() {stack.clear();}
public int getSize() {return stack.getSize();}
public boolean isEmpty() {return stack.isEmpty();}
```



```
public Locator<E> iterator() {return stack.iterator();}
```

The *peek* method returns the object on the top of the stack. The stack is not changed.

```
public E peek() {
    if (stack instanceof Array)
        return stack.get(getSize()-1);
    else
        return stack.get(0);
}
```

Correctness Highlights: If an array-based approach is used then the top of the stack is at the back of the positional collection. Thus, *get* with a position of *size-1* returns the correct answer. If a singly linked list is used, then the top of the stack is at the front of the list, so *get* at position 0 is the correct value. ■

The *push* methods takes *element*, the new element to insert, and inserts it at the top of the stack. It throws an *AtCapacityException* when a bounded stack is full.

```
public void push(E element) {
    if (stack.getSize() == bound)                                //always false when unbounded
        throw new AtCapacityException();
    if (stack instanceof Array)
        stack.addLast(element);
    else
        stack.addFirst(element);
}
```

Correctness Highlights: If an array-based approach is used, the top of the stack is at the back of the array, so calling *addLast(value)* performs the correct mutation. Likewise, when a singly linked list is used, the top of the stack is at the front of the list, so calling *addFirst(value)* performs the correct mutation. ■

The *pop* method removes the item at the top of the stack and returns the removed element. It throws a *NoSuchElementException* when the stack is empty.

```
public E pop() {
    if (stack instanceof Array)
        return stack.removeLast();
    else
        return stack.removeFirst();
}
```

Correctness Highlights: If an array-based approach is used, the top of the stack is at the back of the array. So calling *removeLast* is correct. Likewise, when a singly linked list is used the top of the stack is at the front of the list, so calling *removeFirst* is correct. ■

19.3 Performance Analysis

The constructor takes constant time when a singly linked list is wrapped, and time proportional to the capacity when an array-based representation is used. The asymptotic time complexity of the other methods is as follows:

method	time complexity
constructor	$O(1)$ or $O(n)$
<i>getSize()</i>	$O(1)$
<i>isEmpty()</i>	$O(1)$
<i>iterator()</i>	$O(1)$
<i>peek()</i>	$O(1)$
<i>pop()</i>	$O(1)$
<i>push(o)</i>	$O(1)$
<i>clear()</i>	$O(n)$
<i>contains(o)</i>	$O(n)$
<i>toString()</i>	$O(n)$

Since we have not wrapped the locator, the time complexity for any locator method is the same as that for the wrapped class. Specifically, if an element is removed from the middle of the stack through a locator, it will take constant time if a singly linked list is the underlying representation, and linear time if either a circular array or dynamic circular array is used. If it is important for the application that there be constant time support for mutations through a locator, then a singly linked list should be wrapped, even if the stack need not be tracked. If strict enforcement of the LIFO property is desired, a non-mutating iterator should be used to prevent the application from removing elements from anywhere but the top of the stack.

19.4 Quick Method Reference

Stack Public Methods

- p. 276 *Stack()*
- p. 276 *Stack(int capacity)*
- p. 276 *Stack(int capacity, boolean bounded)*
- p. 276 *Stack(int capacity, boolean bounded, boolean tracked)*
- p. 276 void **clear()**
- p. 276 boolean **contains**(E value)
- p. 276 int **getSize()**
- p. 276 boolean **isEmpty()**
- p. 276 Locator<E> **iterator()**
- p. 277 E **peek()**
- p. 277 E **pop()**
- p. 277 void **push**(E element)

Chapter 20

Set ADT

package collection.set

Collection<*E*>

↳ *Set*<*E*>

A set is an untagged algorithmically positioned collection of elements in which no two elements are equivalent. The primary methods are to add an element, to determine if an element is in the set, and to remove an element. Data structures that implement the Set ADT are designed to perform these methods in constant time. Since elements can be located in constant time, there is no need to support a tracker. In a set, the iteration order can be arbitrary. In fact, the iteration order can change across different iterations. The only requirement is that each persistent element is seen exactly once in a completed iteration.

All of the Set ADT data structures we present are array-based. The array, which we refer to as the **table**, has capacity m . A mechanism is needed to map each potential element of the collection to a slot of the table. To support such a mechanism, Java defines a default *hashCode* method that maps each object to an integer. The designer of each class can override this default *hashCode* to provide a customized behavior. However, it is absolutely necessary that two equivalent objects always have the same value for *hashCode*. In addition, we provide a *Hasher* interface, discussed below, to provide the application the flexibility of defining its own hash code computation over the elements.

In some cases, it is possible to design a *hashCode* method that ensures that all hash codes for the elements of a collection are unique integers in the range $\{0, 1, \dots, m - 1\}$. For example, if the elements that could be placed in the set are the letters from “a” to “z” then one could let $m = 26$ and design a *hashCode* method that maps each letter to a unique integer between 0 and 25. For such applications, there is a unique slot in the table associated with every possible element in the set, which allows for very efficient access.

In general, however, the range of elements that could be held in the set may be very large (or even infinite). Since each hash code is an integer between -2^{31} to $2^{31} - 1$ (inclusive), it is unlikely that two different elements will have the same *hashCode*. However, this cannot be prevented in all cases. Also, since the range of hash codes is typically much larger than m , most set implementations use a technique called **hashing**, where a **hash function** converts the hash code to an integer in $\{0, 1, \dots, m - 1\}$. A well-designed hash function has the property that each element placed in the collection has a $1/m$ probability of hashing to each of the m slots in the *table*.

20.1 Case Study: Airline Travel Agent

Consider a simple automated travel agent that stores all flight information for a designated set of airlines. For each airport served by one of these airlines, information is provided such as the city name, the three letter acronym, the time zone, the latitude, and the longitude for the airport. For each flight the information provided includes the airline name, the originating airport, the departure

time (in the local time zone), the destination airport, the arrival time (in the local time zone), and information about fares.

Suppose the goal is to find the sequence of flights to reach a desired airport as soon as possible from a starting airport A at time t . To support an application to find shortest routes, it is necessary to determine the length of a flight, which in turn requires efficient support to look up the time zone associated with an airport.

We assume that an *Airport* class is defined with instance variables for the city name, three letter acronym, time zone, etc. Since the three letter acronym uniquely identifies a city, it serves as a good field on which to base a comparator to define equivalence between airports. Namely, the comparator will define two airports as equivalent if and only if they have the same acronym.

Using the acronyms to define equivalence, a set provides very efficient support to add a new airport being served, remove an airport no longer served, check if a three letter acronym is associated with one of the airports served, and retrieve the airport instance associated with a given three letter acronym.

In contrast consider the historical event collection case study (Section 1.5), in which a required operation is to return a set of all events in the collection that contain a given word in the event description. To support this efficiently, a mapping needs to be created between each word in the event to the event object. Also, since each word, in general, will appear in more than one event description, it is necessary to map each word to the set of events having that word in their descriptions. Thus, for the historical event collection, a bucket mapping (Chapter 50) is the appropriate ADT.

In the travel agent case study, another option would be to define a mapping from the three letter acronym for each airport, to the airport object. However, such an implementation would not make efficient usage of space since the three letter acronym is already a field within the airport object. For that reason, a set is the better ADT choice. Section 56.1 discusses this case study in more depth.

20.2 Interface

While the *Set* interface is identical to the *Collection* interface, it adds the important restriction that there cannot be any duplicates. Thus any implementation for *Set* must guarantee that no two elements held in the set are equivalent. Implementations for *Collection* to not include this restriction.

It is important that elements in the set are modified only through the interface. In particular, those aspects of the elements that are used to determine equivalence are expected to be immutable, or at least to remain unchanged while the element is in the data structure. For example, if the acronym for an airport would change, then it is essential that the airport object be removed from the set and then reinserted after the acronym is updated.

Critical Mutators for Set: *add* (when it causes resizing), *addAll* (when it causes resizing), *ensureCapacity*, *remove*, *retainAll*, *trimToSize*

20.3 Terminology

In addition to the definitions given in Chapter 7, we use the following definitions and conventions throughout our discussion of the Set ADT and the data structures that implement it.

- Let $\mathcal{U}$ be the universe of elements that could be placed in the collection. For example, if a set holds some subset of the states in the United States, then $\mathcal{U} = \{\text{Alabama, Alaska, } \dots, \text{Wyoming}\}$ and $|\mathcal{U}| = 50$.
- The size of *table* is m . That is, $m = \text{table.length}$.
- When a hash function is used, *table* is called the **hash table**.
- We refer to $\text{table}[i]$ as **slot** i of the hash table.
- When no hash function is needed, for ease of exposition we treat the hash function as if it were the **identity function** where $\text{hash}(i) = i$. In the implementation for such cases, the hash code is directly used instead of computing the hash of the hash code.
- When the hash code of an element is i , we say that element e **hashes to** slot i .
- A **collision** occurs between elements $e1$ and $e2$ if both hash to the same slot.
- We say slot i is **in use** when slot i refers to an element currently in the collection (or a list holding an element in the collection).
- As discussed in more depth in Chapter 22, when an element is removed in open addressing, the slot currently holding the element is marked as deleted. When this occurs we say that slot i is a **deleted slot**.
- Any slot that is neither in use nor a deleted slot is called an **empty slot**. The distinction between an empty slot and deleted slot is important.
- We let $\alpha = (n + d)/m$ denote the **load** of a hash table, where d is the number of deleted slots. With the exception of the open addressing data structure, $d = 0$.
- Let S be a set, and let *compare* be the equivalence tester for S . We say that elements $e1$ and $e2$ are **equivalent with respect to** S if and only if $\text{compare}(e1, e2)$ returns 0. When S is clear from context, we simply say that elements $e1$ and $e2$ in set S are **equivalent** (written as $e1 \cong e2$), when they are equivalent with respect to S .
- Set S is said to **contain** $e1$ whenever there is some $e2$ in S for which $e1 \cong e2$.
- We say that elements $e1$ and $e2$ are **distinct** (denoted $e1 \not\cong e2$) when $e1$ and $e2$ are not equivalent with respect to S .
- In a **successful search** for target element e , an equivalent element is found. In an **unsuccessful search** for target element e , no equivalent element is found.

20.4 Hasher Interface

package collection.set

Hasher<*E*>

To provide flexibility for computing the hash code for elements, we introduce the following *Hasher* interface. By providing a hasher object to the constructor of a set, the application can

specify how the hash codes are to be computed over the elements of the set. This means that one need not rely on the *hashCode* methods defined within the elements themselves.

The method *getHashCode* takes as its parameter *element*, the element for which the hash code should be computed. It returns the hash code for the given element, which may not necessarily be related to the *element.hashCode*.

```
public int getHashCode(E element);
```

As will be discussed for each set implementation, the desired size of the hash table depends upon both the hashing algorithm and the way hash codes are computed. Consequently, the *Hasher* interface includes a method *getTableSize* that takes as its parameters *capacity*, the desired capacity of the collection and *load*, the desired load factor of the table and returns the proper table size in accordance with the hashing algorithm and the way hash codes are computed.

```
public int getTableSize(int capacity, double load);
```

20.5 Competing ADTs

We briefly discuss ADTs that may be appropriate in a situation when a Set ADT is being considered. This discussion assumes familiarity with the difference between tagged and untagged collections as discussed in Section 2.6.2.

Mapping ADT: If there is a unique tag (i.e., a key) distinct from the element that the application program uses to retrieve the element, then the Mapping ADT should be considered. In contrast, when the information being used for the tag is contained within the elements themselves, then a Set ADT (with a comparator that uses the relevant information to define equivalence between elements) will be more space efficient.

BucketMapping ADT: If the application requires that equivalent elements be inserted and grouped together, then the BucketMapping ADT is likely to be the right choice.

Hierarchy of Mappings: If the number of elements to be held in the collection is so large that their references will not all fit into main memory, then a hierarchy of mappings should be used.

DigitizedOrderedCollection ADT: While a digitized ordered collection is most commonly used when there is an ordering defined among the elements, it can also be used to implement a set. If the elements can be viewed as a sequence of digits, and elements are often distinguishable after looking at relatively few digits (as is often the case with an unsuccessful search), then the DigitizedOrderedCollection ADT is more appropriate than a set. For elements that are naturally defined as a sequence of digits (e.g., words), using a digitized ordered collection avoids the need to make potentially expensive comparisons among the elements since a digitizer makes use of a canonical ordering of the digits. In particular, a digitized ordered collection can be more efficient than a Set ADT when a constant number of comparisons between elements is more expensive than performing *getDigit* for the number of digits that must typically be examined to distinguish the desired elements from all other elements in the collection.

20.6 Selecting a Data Structure

There are three efficient implementations for the Set ADT: direct addressing, separate chaining, and open addressing. All of these data structures can locate an element in expected constant time, so there is little reason to incur the extra costs associated with a tracker. Thus, for the Set ADT, we only provide untracked implementations. However, if desired, a tracked implementation could easily be implemented.

All three of these data structures assume that the elements to be maintained are distinct. If the application program calls *add(e1)* when there is an *e2* in the set with $e1 \cong e2$, *e1* replaces *e2* in the set. The differences between the three data structures concern what happens when more than one distinct element hashes to the same slot.

Direct addressing requires that no two elements in $\mathcal{U}$ hash to the same slot, thus preventing collisions. However, such an approach requires the table size be at least $|\mathcal{U}|$, which is often not feasible.

Separate chaining places all elements that hash to the same slot in a list.

Open addressing defines a **probe sequence**, which is a permutation of all available slots. The probe sequence is determined separately for each element by using a **secondary hash function** that specifies how many slots to move forward when discovering a slot is in use. The original hash function is called the **primary hash function**. Whenever a collision occurs based on the primary hash function, open addressing moves to successive slots in the probe sequence, based on the secondary hash function, until finding an available slot. the probe sequence.

To select among these data structures, consider the following questions.

Will a large fraction of elements in $|\mathcal{U}|$ be held in the set? When $n \geq |\mathcal{U}|/4$ then it is reasonable to allocate the table large enough so that each element in $|\mathcal{U}|$ can be associated with a unique slot of the table, ensuring there are no collisions. This is the approach taken by direct addressing.

Are deletions frequent? In open addressing, the time complexity of methods that must locate an element depends on both n and d . The complexity of locating an element in the other data structures depends only on n . So if deletions are frequent, open addressing probably should be avoided.

Does the data structure need to minimize space usage? There is generally a trade-off between space usage and time complexity. Differences in the collision resolution methods between separate chaining and open addressing allow open addressing to minimize space usage.

Does n have periods of significant change? As in the dynamic array (Chapter 13), as n changes significantly, the hash table must be resized. With direct addressing resizing cannot occur since the table size is $|\mathcal{U}|$, and it is assumed that $\mathcal{U}$ does not change. Because of the way collisions are handled, separate chaining can resize the table less frequently than open addressing. Neither will automatically resize the table if the user calls *ensureCapacity* appropriately.

The answer to the first question is all that is needed to determine if direct addressing should be used. Direct addressing provides the most efficient implementation for all the methods in the Set

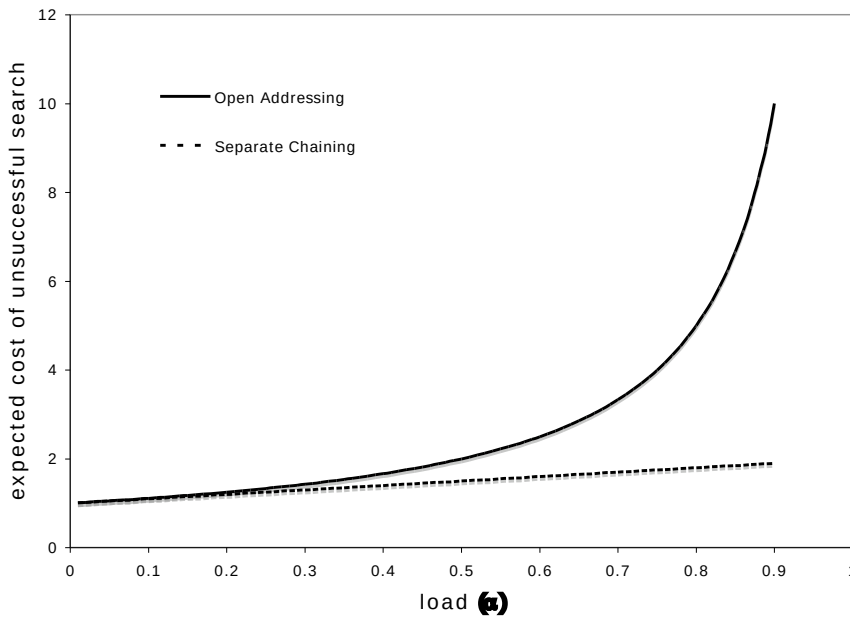


Figure 20.1

A comparison of the relationship of the expected number of probes for an unsuccessful search as a function of the load. It is important to remember that for the same amount of space, open addressing provides a lower value of α than separate chaining.

interface. The drawback of direct addressing is that the size of the table must be $|\mathcal{U}|$. When n is significantly smaller than $|\mathcal{U}|/4$ then either separate chaining or open addressing should be used since they can perform nearly as well with space usage of roughly $4n$.

The choice between open addressing and separate chaining is much more subtle. One very important distinction is that for open addressing, some removed elements still logically occupy slots of the hash table and therefore increase the expected time of locating a desired element. (See [Chapter 22](#) for an in-depth discussion about this issue.) Recall that we use d to denote the number of hash table slots that are occupied due to objects that have been removed from the collection. For directed addressing and separate chaining, d is always 0 since deleted elements do not leave any remnants behind to affect future performance.

For both separate chaining and open addressing, the complexities of all the primary methods depend on either the cost of a successful or unsuccessful search. In order to ensure that all elements in the set are distinct, an unsuccessful search must be performed prior to each insertion. Furthermore, most of the computation required when inserting a new element is that of the unsuccessful search. We define a **probe** as a slot/pointer that must be examined when searching for a specified item in the set. Since the computation cost for the *contains*, *add*, and *remove* methods are proportional to the number of probes, in making our comparisons between these two data structures, we compare the expected number of probes.

For both open addressing and separate chaining, there is a trade-off between the space usage (i.e., the table size) and the computation cost which is directly related to the load. The application program can specify a target load (via the constructor or the *setTargetLoad* method), which dictates the size to use for the hash table. The smaller the desired load, the faster the methods will run, but the more memory the hash table will occupy.

For open addressing, the expected number of probes for an unsuccessful search is $\frac{1}{1-\alpha} = 1 + \alpha + \alpha^2 + \alpha^3 + \dots$, which approaches infinity as α approaches 1. The expected number of probes for a

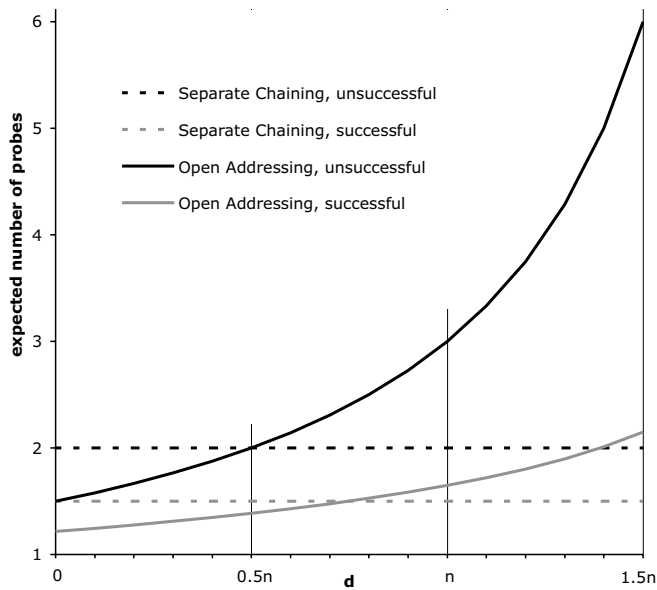


Figure 20.2

A comparison of the relationship of the expected number of probes for separate chaining and open addressing when they have roughly the same space usage. (Namely, $m_{SC} = n$ and $m_{OA} = 3n$.) We consider as d ranges from 0 to $3n/2$.

successful search is $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$. The performance degrades so badly as α approaches 1, that typically one must ensure that $\alpha \leq 0.75$. Selecting $\alpha = 0.5$ provides a good trade-off between the time and space complexity.

For *separate chaining*, the expected number of probes for an unsuccessful search is $1 + \alpha$, and the expected number of probes for a successful search is $1 + \frac{\alpha}{2} - \frac{\alpha}{2n}$. Selecting $\alpha = 1.0$ provides a good trade-off between the time and space complexity.* With separate chaining, there is no strict upper bound for α .

Figure 20.1 compares the expected number of probes for an unsuccessful search as a function of the load for both open addressing and separate chaining. Observe that for a load factor of 3 for separate chaining gives the same expected number of probes for an unsuccessful search as a load factor of 0.75 does for open addressing.

To illustrate the trade-offs between separate chaining and open addressing, it helps to consider their computational complexity when both have roughly the same space usage. For ease of exposition, we assume that the internal representation for open addressing takes space $m = \text{table.length}$ and ignore instance variables such as *size* since they do not contribute significantly to the space usage. As discussed in Section 23.7, for separate chaining, the space usage is approximately $m + 2n$ since each item in the chain requires two references. (For any object-oriented language with polymorphism, the compiler must maintain the type of the object to dispatch the method calls. Java makes this information accessible through the method *getClass* that can be called on any object. In any such language, the space usage is really $m + 3n$ since there is a third hidden reference for each chain item. If desired, the analysis could be repeated with this higher space usage for separate chaining. Our analysis here illustrates that even with the conservative value of $m + 2n$ the space usage for separate chaining is high as compared to open addressing.) A good choice for the load when using

*The documentation provided with Java recommends a load factor of 0.75. See Table 20.4 for a comparison of the trade-offs for a variety of load factors including 0.75 and 1.0.

data structure	unsuccessful search	successful search
<i>Separate Chaining</i> , $\alpha = 1$	2	$\frac{3}{2} - \frac{1}{2n}$
<i>Open Addressing</i> , $\alpha = 1/3$	$\frac{3n}{2n - d}$	$\frac{3n}{n + d} \ln \frac{3n}{2n - d}$

Figure 20.3

A comparison of the expected number of probes for separate chaining and open addressing when they both have a space usage of approximately $4n$. This corresponds to having $m_{SC} = n$ and $m_{OA} = 3n$.

separate chaining is $\alpha = 1$. To achieve this load, we select a hash table size of $m_{SC} = n$. For this hash table size, the space usage is approximately $3n$. To make open addressing roughly match the space usage of separate chaining, we set $m_{OA} = 3n$ which corresponds to a load $\alpha = 1/3$ (when $d = 0$). For these choices for the load factor, the expected number of probes is shown in Table 20.6.

Figure 20.2 provides a comparison between separate chaining and open addressing for $m_{SC} = n$ and $m_{OA} = 3n$ as d ranges from 0 to $n/2$. Thus for a fixed space usage, open addressing leads to fewer probes with these load factors even when d is as large as $n/2$.

Since the time complexity for open addressing is a fairly complex formula, it is hard to make an intuitive comparison. Although resizing of *table* cannot be done very frequently since it takes linear time to do so, mathematically more intuitive formulas are obtained by assuming that one has an estimate for d , and sizes the open addressing hash table accordingly (using *setTargetLoad*). Doing this analysis results in the comparison shown in Table 20.4.

20.7 Summary of Set Data Structures

Table 20.5 provides a high-level summary of the trade-offs between the data structures we present for the Set ADT. For all three data structures we assume that the space usage is roughly $4n$ (or less than $4n$ for direct addressing). In Table 20.5, we define excellent performance as worst-case constant time, fair performance as expected-case constant time, and poor performance as linear time. Figure 20.6 shows the class hierarchy for the data structures presented in this book for the Set ADT.

DirectAddressing: This data structure provides excellent performance, but $O(|\mathcal{U}|)$ space is required. This data structure is the best choice when at least a quarter of the elements in $\mathcal{U}$ are expected to be held in the collection. When n is expected to be much smaller than $|\mathcal{U}|/4$, then either separate chaining or open addressing should be used.

SeparateChaining: This data structure should be considered when only a small fraction of the elements in $\mathcal{U}$ will be stored in the collection. It is the right choice in this setting when deletions occur frequently, or if n changes dramatically since it can be resized relatively infrequently. In the provided implementation, it is only resized when n grows 4-fold, which keeps the expected load within a factor of two of the desired value. If the space usage can vary even more, then without much degradation in performance, the implementation can be modified to resize less frequently. However, the space usage required for separate chaining is higher than that for the other data structures.

Data Structure	Unsuccessful Search	Successful Search	Approximate Space Usage
Direct Addressing	1	1	m
Separate Chaining	$1 + \alpha$	$1 + \frac{\alpha}{2} - \frac{\alpha}{2n}$	$2n + m = n \left(3 + \frac{1}{\alpha} \right)$
Separate Chaining ($\alpha=1/2$)	1.5	≈ 1.25	$4n$
Separate Chaining ($\alpha=3/4$)	1.75	≈ 1.375	$3\frac{1}{3}n$
Separate Chaining ($\alpha=1$)	2	≈ 1.5	$3n$
Separate Chaining ($\alpha=3$)	4	≈ 2.5	$2\frac{1}{3}n$
Open Addressing	$\frac{1}{1 - \alpha}$	$\frac{1}{\alpha} \ln \frac{1}{1 - \alpha}$	$m = \frac{(n + d)}{\alpha}$
Open Addressing ($\alpha=1/4$)	$\frac{4}{3}n$	≈ 1.15	$4(n + d)$
Open Addressing ($\alpha=1/2$)	2	≈ 1.39	$2(n + d)$
Open Addressing ($\alpha=3/4$)	4	≈ 1.85	$\frac{4}{3}(n + d)$

Table 20.4 A comparison between all three mapping data structures. For separate chaining, the recommended value for α is 1.0, and with open addressing the recommended value for α is 0.5.

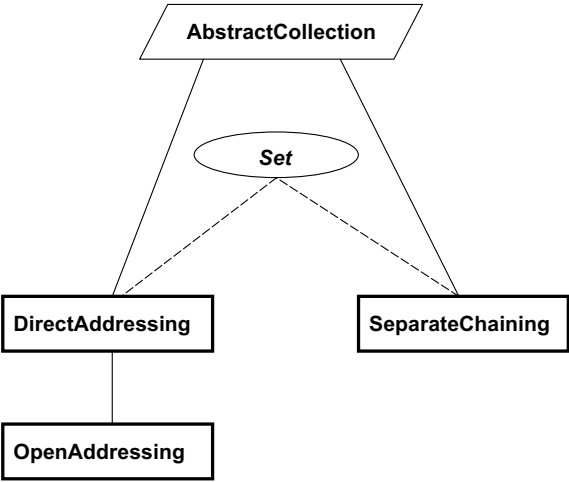


Figure 20.6
The class hierarchy for the Set ADT data structures.

Key				
⦿	Excellent			
○	Very Good			
●	Fair			
•	Method does nothing			
Method		DirectAddressing	SeparateChaining	OpenAddressing
Set Methods	add(o), addAll(c), per element	⦿	○	○
	clear(), per element	⦿	⦿	⦿
	contains(o)	⦿	○	○
	ensureCapacity(x)	•	●	●
	getEquivalentElement(o)	⦿	○	○
	getLocator(o)	⦿	○	○
	remove(o)	⦿	○	○
	retainAll(c), per element	⦿	○	○
Other Issues	accept(v), toArray(), toString(), per element	⦿	⦿	⦿
	trimToSize(), per element	•	⦿	⦿
	space usage	⦿	○	⦿
	space usage independent of universe size		✓	✓
Marker Methods	how remove affects future operations	⦿	⦿	○
	frequency of resizing	•	⦿	○
	advance()	○	○	○
	get()	○	○	○
	hasNext()	○	○	○
	inCollection()	○	⦿	○
	next()	○	○	○
	remove()	○	○	○
	retreat()	○	○	○
Time Complexity		Other Issues		
⦿	worst-case O(1) time	⦿	excellent	
○	expected-case O(1) time	○	fair	
●	O(n) time	•	not applicable	
•	method does nothing			

Table 20.5 Trade-offs among the data structures that implement the Set ADT. For all three we assume the space usage is at most $4n$ which requires that for direct addressing that $|\mathcal{U}| \leq 3n$.

OpenAddressing: This data structure should also be considered when only a small fraction of the elements in $\mathcal{U}$ will be stored in the collection. As compared with separate chaining, for the same space usage, the primary methods generally run faster. However, the computation costs of the primary methods grow as the number of deletions grows. If deletions are frequent, then separate chaining (or if applicable, direct addressing) should be considered. Also, as compared with separate chaining, resizing must be performed more frequently if the capacity provided in the constructor (or via *ensureCapacity*) is not adequate.

Although there are subtle differences between separate chaining and open addressing, asymptotically their performance is the same.

20.8 Further Reading

Knuth [97] is a good reference for hashing and its analysis. Knuth credits H.P. Luhn for first presenting hash tables, and the chaining method for resolving collisions, in an internal IBM memorandum of 1953. Peterson [123] first describes open addressing. He credits A.L. Samuel, G.M. Amdahl, and E. Boehme for devising open addressing in 1954. G.M. Amdahl organized the idea of linear probing. Robert Morris [117] first proposed double hashing, which is the collision technique we use for open addressing.

We have focused only on hashing for the dynamic setting in which it is expected that elements are added and removed from the set over time. There are some applications that are naturally **static** in that the set of elements does not change. **Perfect hashing** is designed to guarantee a worst-case constant time search for the static setting. Fredman et al. [62] developed perfect hashing for static sets, and later Dietzfelbinger et al. [45] extended perfect hashing for dynamic sets.

Chapter 21

Direct Addressing Data Structure

package `collection.set`

`AbstractCollection<E>` implements *Collection<E>*

└ `DirectAddressing<E>` implements *Set<E>*

Uses: Java primitive array

Used By: `DirectAddressingMapping` (Section 49.7.1)

Strengths: All primary methods run very quickly, with most involving only a call to the *hashCode* method and a single array access. Also, for this data structure there are no critical mutators. That is, iteration can always continue with a consistent order in spite of concurrent modifications.

Weaknesses: This data structure requires that $|\mathcal{U}| = m$ and that *hashCode* is a one-to-one function from the elements of $\mathcal{U}$ to $\{0, 1, \dots, m - 1\}$. Also, this data structure uses an array of size $|\mathcal{U}|$ as the internal representation. Such space usage is excessive unless n is roughly $|\mathcal{U}|/4$ or greater.

Critical Mutators: none

Competing Data Structures: For most situations, $|\mathcal{U}|$ is significantly larger than n . For example, consider the set of registered voters in a county with 500,000 eligible voters, where each voter is identified by a social security number. Since the social security number is a 9-digit number, $|\mathcal{U}| = 10^{10}$. Even if such an array could be allocated, it is a very inefficient use of space to maintain a set of at most 500,000 voters. In such situations, unless space is to be sacrificed to guarantee very fast worst-case performance, open addressing (Chapter 22) or separate chaining (Chapter 23) should be considered.

21.1 Internal Representation

For this data structure, the hash function is the identity function, so each possible element maps to a unique slot of the table. For the sake of efficiency, we use *e.hashCode()* directly in place of *hash(e.hashCode())*. The elements in the set are stored in an underlying Java primitive array, *table*, that has capacity $m = |\mathcal{U}|$. Let $\mathcal{U} = \{x_0, x_1, \dots, x_{m-1}\}$ where $x_i.hashCode() = i$. When x_i is in the set, *table[i]* holds a reference to x_i . Otherwise, *table[i]* = *null*. To determine set membership, add a new element, or remove an existing element, the only computation required is that performed by *hashCode*, followed by a single array access or mutation.

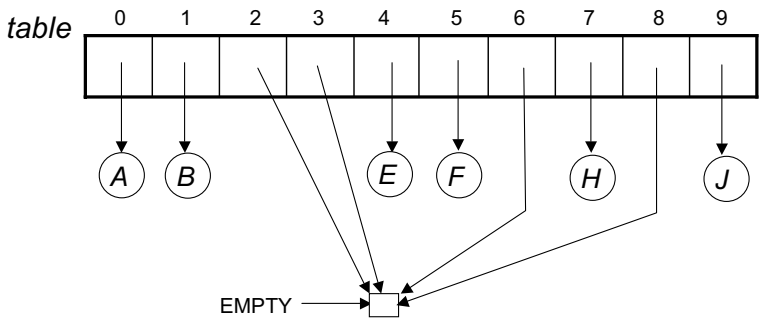


Figure 21.1
A populated example for the set $\{A, B, E, F, H, J\}$ where $\mathcal{U}$ contains letters from A to J (inclusive), and the hash code for the i^{th} letter of the alphabet is $i - 1$.

Instance Variables and Constants: The variable *table* is the underlying array, with one slot for each possible element. Variables *size*, *DEFAULT_CAPACITY*, *FORE*, *NOT_FOUND*, *version*, and *comp* are inherited from *AbstractCollection*. Since iteration proceeds by going from slot 0 to *table.size* - 1, the code for the locator is simplified by logically viewing slot -1 as *FORE* and slot *table.length* as *AFT*. The singleton *Objects.EMPTY* (abbreviated as *EMPTY*) is referenced by a slot that is not in use. By definition $EMPTY \notin \mathcal{U}$. As discussed in Section 5.2.1, this is used instead of *null*, since *null* may be an element of $\mathcal{U}$.

```
Hasher<? super E> hasher;           //computes hash codes
Object[] table;                       //the underlying array
```

The capacity of the collection is *table.length*.

Populated Example: Figure 21.1 shows the internal representation for a set containing $\{A, B, E, F, H, J\}$ where $\mathcal{U}$ is the set of letters from A to J , and the hash code for the i^{th} letter of the alphabet is $i - 1$. That is $A.hashCode() = 0$, $B.hashCode = 1$, etc. An arrow is used to denote a non-*null* pointer to an element. For the set illustrated in Figure 21.1, *size* is 6 and $m = |\mathcal{U}| = 10$.

Abstraction Function: Let *DA* be a direct addressing instance. The abstraction function

$$AF(DA) = \{v \in \mathcal{U} \mid \exists i \text{ for which } table[i] = v\}.$$

Terminology: When slot i is not in use, we define *prev*(i) to be the largest integer j such that $j < i$, and slot j is in use. If there is no such j , then by definition *prev*(i) = *FORE*.

Optimizations: To allow open addressing to extend direct addressing, some extra methods such as *slotInUse*, and *clearSlot* are added, which introduce a small amount of overhead. Also, each marker must include the maintenance and checking of the version number, even though direct addressing has no critical mutators. These could be removed in an optimized version of direct addressing that did not have open addressing as a subclass.

If for all $e \in \mathcal{U}$, e is an integer and $e.hashCode() = e$, then *table* could be a bit vector in which $table[e] = \text{true}$ when e is in the set and $table[e] = \text{false}$ when e is not in the set. Also, in this case, the *hashCode*() function would not be needed.

Similarly, if iteration is not required, then it is not necessary to retain references to the elements of $\mathcal{U}$, and a bit vector suffices to implement the *add*, *remove*, and *contain* methods.

21.2 Representation Properties

Along with `SIZE`, which is inherited from `AbstractCollection`, we introduce the following two representation property. Recall that this data structure requires that for any two distinct elements $e1 \in \mathcal{U}$ and $e2 \in \mathcal{U}$, $hashCode(e1) = hashCode(e2)$. The correctness of many methods rely on this requirement.

PLACEMENT: An element $e \in \mathcal{U}$ is in this set if and only if $table[e.hashCode()] \cong e$. Thus, if $table[e.hashCode()]$ is empty, then there is no element equivalent to e in this set.

Together these properties imply that for a set S , $e \in S$ if and only if $table[e.hashCode()] \cong e$.

21.3 Default Direct Addressing Hasher Class

The following direct addressing hasher allocates slot 0 for the *null* element and adds one to each of the naturally computed hash codes for the elements to compensate. Similarly, the table size is computed to be one greater than the given capacity.

```
public static class DirectAddressingHasher<E> implements Hasher<E> {
    public int getHashCode(E element) {
        return (element == null) ? 0 : 1+element.hashCode();
    }
    public int getTableSize(int capacity, double load) {
        return capacity + 1;
    }
}
```

21.4 Methods

In this section we present the internal and public methods for the `DirectAddressing` class.

21.4.1 Constructors

The most general constructor takes three arguments: *capacity*, the desired capacity, *equivalenceTester*, the comparator that defines equivalence of objects, and *hasher*, the user-supplied hash code computation. It creates a hash table with the specified capacity (plus one to handle *null* values), and with the given comparator and hasher. It is expected that $capacity = |\mathcal{U}|$. It throws an *IllegalArgumentException* when $capacity < 0$. Since the elements held in the mapping could be *null*, *EMPTY* is used instead of *null* for initialization.

In directed addressing the equivalence tester is not used, since there is a requirement that two elements are equivalent if and only if they have the same *hashCode*. However, the subclass `OpenAddressing` must provide this added parameter.


```

protected DirectAddressing(int capacity, double load,
    Comparator<? super E> equivalenceTester, Hasher<? super E> hasher) {
    super(equivalenceTester);
    this.hasher = hasher;
    table = new Object[hasher.getTableSize(capacity, load)];
    Arrays.fill(table, Objects.EMPTY);
    size = 0;
}

```

Correctness Highlights: SIZE is satisfied since $n = size = 0$. Since $S = \emptyset$, setting all slots to *EMPTY* satisfies PLACEMENT. ■

As a convenience, the following constructor takes two arguments: *capacity*, the desired capacity, and *comp*, the comparator that defines equivalence of objects. It creates a hash table with the specified capacity (plus one to handle *null* values), and with the given comparator and default direct addressing hasher. It throws an *IllegalArgumentException* when *capacity* < 0.

```

protected DirectAddressing(int capacity, Comparator<? super E> comp) {
    this(capacity, 1, comp, new DirectAddressingHasher<E>());
}

```

We also include the following constructor that takes, *capacity*, the desired capacity. It creates a hash table with the specified capacity (plus one to handle *null* values), and with the default comparator and direct addressing hasher. It throws an *IllegalArgumentException* when *capacity* < 0. It is expected that *capacity* = $|\mathcal{U}|$. Since the elements held in the mapping could be *null*, *EMPTY* is used instead of *null* for initialization.

```

public DirectAddressing(int capacity) {
    this(capacity, Objects.DEFAULT_EQUIVALENCE_TESTER);
}

```

21.4.2 Trivial Accessors

The *isEmpty* and *getSize* methods are inherited from *AbstractCollection*. Recall that *getCapacity* returns the capacity of the collection (which is $|\mathcal{U}|$).

```

public int getCapacity() {
    return table.length;
}

```

Correctness Highlights: By definition, the number of elements that can be stored in the set is $|\mathcal{U}|$ which is equal to *table.length*. ■

21.4.3 Representation Accessors

To prepare for the extension of direct addressing to open addressing, the representation accessors *inUse* is introduced. This method takes *slot*, a valid slot of the table, and returns *true* if and only

if *slot* references an element of the set. It throws an *IllegalHashCodeException* when *slot* is not a valid slot. Since any method that accesses the table, uses this method, all public methods will throw an *IllegalHashCodeException* when the hash code for an element does not correspond to a valid slot.

```

boolean inUse(int slot){
    try {
        return table[slot] != Objects.EMPTY;
    } catch (ArrayIndexOutOfBoundsException exc){
        throw new IllegalHashCodeException(slot);
    }
}

```

Correctness Highlights: By PLACEMENT, if *slot* is in $\{0, 1, \dots, \text{table.length} - 1\}$, then the correct answer is returned. Otherwise, an *IllegalHashCodeException* is properly thrown since direct addressing requires that the hash code to be an integer between 0 and $m - 1$ which exactly corresponds to the valid indices of *table*. ■

21.4.4 Algorithmic Accessors

The internal method used most often is one to find the slot in the table that holds a desired element. The *locate* method takes *element*, the target, and returns the slot number in the table where *element* is held, or the constant *NOT_FOUND* if *element*. It throws an *IllegalHashCodeException* when *hashCode* is not a valid slot.

```

int locate(E element) {
    return hasher.getHashCode(element);
}

```

Correctness Highlights: The correctness follows from PLACEMENT. ■

The method *contains* takes *element*, the target, and returns *true* if and only if an equivalent element exists in the collection.

```

public boolean contains(E element) {
    return inUse(locate(element));
}

```

Correctness Highlights: Follows from the correctness of *locate* and the correctness of *inUse*. ■

The *getEquivalentElement* method takes *target*, the target element, and returns an equivalent element that is in the collection. It throws a *NoSuchElementException* when there is no equivalent element in the collection.

```

public E getEquivalentElement(E target) {
    int slot = locate(target);
    if (!inUse(slot))
        throw new NoSuchElementException();
}

```

```

    else
        return (E) table[slot];
}

```

Correctness Highlights: Follows from that of PLACEMENT and the correctness of *locate*. ■

21.4.5 Representation Mutators

Since direct addressing requires that the capacity is $|\mathcal{U}|$, it is not meaningful to modify the capacity unless $\mathcal{U}$ changes. If $\mathcal{U}$ were to change then a new set should be created, and all elements should be inserted into that set. Thus, the public method *ensureCapacity* throws an *UnsupportedOperationException* when it is called for direct addressing. An alternative would be to have this method do nothing, but that option would be more likely to result in surprising hash code out of bounds exceptions later in the execution.

```

public void ensureCapacity(int capacity) {
    throw new UnsupportedOperationException();
}

```

Likewise, the method *trimToSize* throws an *UnsupportedOperationException* when it is called for direct addressing. If the application needs a compact representation of the set, the *toArray* method returns an array of size n .

```

public void trimToSize() {
    throw new UnsupportedOperationException();
}

```

21.4.6 Content Mutators

Methods to Perform Insertion

The *add* method takes *element*, the element to add to the set. If an equivalent element x is in the set, then *element* replaces x . The method throws an *IllegalHashCodeException* when *hashCode* is not a valid slot in the table.

```

public void add(E element) {
    int slot = locate(element);
    if (!inUse(slot))
        size++;
    table[slot] = element;
}

```

Correctness Highlights: Recall that when an equivalent element is currently in the set, the semantics of *add* is to replace the existing element by the new element. However, when replacing an existing element by an equivalent element, the size of the set does not change. Thus, SIZE is preserved by incrementing *size* exactly when the slot corresponding to *element* is not in use. The rest of the correctness follows from PLACEMENT, which is easily seen to be preserved. ■

Methods to Perform Deletion

To support extending this class for the open addressing data structure, the representation mutator *clearSlot* is introduced. This method takes *slot*, a valid slot number, and removes the element (if any) held in that slot from this set. The method returns *true* if the slot was in use. It throws an *IllegalHashCodeException* when *slot* is not a valid slot.

```

boolean clearSlot(int slot){
    if (inUse(slot)) {
        table[slot] = Objects.EMPTY;
        size--;
        return true;
    }
    return false;
}

```

Correctness Highlights: By PLACEMENT, setting *table[slot]* to *EMPTY* removes the element (if any) held in *slot*. The exception is appropriately thrown since the valid slots exactly correspond to the indices of *table*. ■

The *remove* method takes *element*, the element (if it exists) to be remove from the set. It returns *true* if and only if the element had been in the set and was therefore removed.

```

public boolean remove(E element) {
    return clearSlot(locate(element));
}

```

Correctness Highlights: By the correctness of *locate*, the local variable *slot* contains the slot in the table where the element is located (or has the value *NOT_IN_USE* if the element is not in the Set). The *clearSlot* method preserves PLACEMENT, and the decrement of *size* in *clearSlot* preserves SIZE. ■

The *clear* method removes all elements from the collection.

```

public void clear() {
    Arrays.fill(table, Objects.EMPTY);
    size = 0;
}

```

Correctness Highlights: PLACEMENT is maintained by setting all slots to *EMPTY*, and SIZE is maintained by setting *size* to 0. ■

21.4.7 Locator Initializers

The *iterator* method creates a new marker that is placed at FORE.

```

public Locator<E> iterator() {
    return new Marker(FORE);
}

```

Correctness Highlights: Follows from the Marker constructor and the fact that FORE logically precedes the slot 0 of the table. ■

The *getLocator* method takes *value*, the target, and returns a marker initialized at an equivalent element in the set. It throws a *NoSuchElementException* when there is no equivalent element in this set.

```
public Locator<E> getLocator(E value) {
    int location = locate(value);
    if (!inUse(location))
        throw new NoSuchElementException();
    return new Marker(location);
}
```

Correctness Highlights: Follows from *locate* and the Marker class constructor. ■

21.5 Marker Inner Class

`AbstractCollection<E>.AbstractLocator<E>` implements *Locator*

↳ **Marker** implements *Locator<E>*

Recall that a marker simply marks a location within a data structure, enabling a user application to iterate through a data structure possibly making mutations relative to the current marker location. Since direct addressing supports fast worst-case constant time access for an element, there is no need to support a tracker.

Each marker includes the instance variable *slot* that stores the slot of the table where the marker is located. Recall that we view FORE as being at slot “-1” and AFT as being at slot “*m*” where $m = \text{table.length}$.

```
private int slot; //marker location
```

For direct addressing, the slot number of the hash table is used as the location that is marked. The iteration order for direct addressing has no relation to the order in which the elements were added to the set. Iteration proceeds from slot 0 to slot $m - 1$. More formally, let $s_0, \dots, s_{n-1}$ be the slots that are in use where $s_0 < s_1 < \dots < s_{n-1}$. The iteration order that is defined by the marker is $\langle s_0, s_1, \dots, s_{n-1} \rangle$. Therefore, the iteration order is defined by the relative order of the hash code values, so when that order has a consistent semantics, direct addressing can be viewed as an ordered collection (Chapter 29).

Recall that when an element is removed, any markers at the removed element are logically at the previous element in the iteration order, so a subsequent call to *advance* moves to the next unseen item in the iteration order. While the implementation could directly perform this update for the marker when it calls *remove*, that could waste time. Consider the populated example of [Figure 21.1](#), and suppose that $loc = 4$. When *loc.remove()* is called, then the marker is logically at slot 1. However, the following method call may be *loc.advance()*, in which case unnecessary computation

would have been performed in determining that $prev(4) = 1$. Hence, our implementation physically leaves the marker at the current slot when an item is removed. However, logically, the marker holding i is at $prev(i)$. Another advantage of this approach is that any marker at a removed element (even if *remove* was not called through the marker), has the same behavior, which allows iteration to properly proceed through any active markers even after an element has been removed. We introduce an abstraction function to capture this behavior.

Abstraction Function: Let *loc* be a marker instance. The abstraction function for the marker is as follows:

$$AF(loc) = \begin{cases} loc.slot & loc.slot \text{ is in use} \\ prev(loc.slot) & \text{otherwise} \end{cases}$$

In our correctness arguments, we argue that this abstraction function is consistent with the iteration order given above and the semantics required of the marker as described in Section 5.8.

The Marker constructor takes *slot*, the slot to store within the marker. This constructor throws an *IllegalArgumentException* when *slot* is not FORE (-1), AFT (*m*), or a valid slot of the table. While direct addressing never invalidates any markers, OpenAddressing, which directly uses this marker class, will invalidate all active markers when the table is resized. Thus, the version number is included here, and validity checks are performed.

```
Marker(int slot) {
    if (((slot < -1) || (slot ≥ table.length)))
        throw new IllegalArgumentException("slot is " + slot);
    this.slot = slot;
    versionNumber = version.getCount();
}
```

The marker also provides an accessor, *inCollection*, that returns *true* if and only if the marker is at an element of the collection. This method also updates *slot* so that it is either FORE, AFT, or a valid slot that is in use. We use this fact in all methods that call *inCollection()*.

```
public boolean inCollection() throws ConcurrentModificationException {
    checkValidity();
    if (slot != FORE && slot != table.length && !inUse(slot))
        retreat();
    return (slot != FORE && slot != table.length);
}
```

Correctness Highlights: Recall that when an element is removed, the abstraction function specifies that the marker location is logically at $pred(slot)$, yet physically no change is made to *slot*. By the correctness of *retreat* and *inUse*, when the marker is at a slot of the table that is not in use, it will be updated to $pred(slot)$. The rest of the correctness follows from the semantics of FORE and AFT. ■

The marker provides an accessor, *get*, that returns the element stored at the marker location. It throws a *NoSuchElementException* when the marker is logically at FORE or AFT.

```
public E get() throws ConcurrentModificationException {
    if (!inCollection())
        throw new NoSuchElementException();
    return (E) table[slot];
}
```

Correctness Highlights: Follows from the correctness of *inCollection*, and *PLACEMENT*, which guarantees that *table[slot]* holds the desired element. ■

The *advance* method moves the marker to the next slot in use and returns *true* if and only if there is an element at the updated marker location. It throws an *AtBoundaryException* when the marker is at AFT since there is no place to advance.

```
public boolean advance() throws ConcurrentModificationException {
    if (slot == table.length)
        throw new AtBoundaryException("Already after end.");
    checkValidity();
    while (++slot < table.length)           //slot will be at AFT when loop exits
        if (inUse(slot))
            return true;
    return false;
}
```

Correctness Highlights: If the current marker location is AFT then an exception is correctly thrown. Since AFT is -1, in all other cases, the marker should be moved to the next slot in use starting with *slot + 1*. Observe that even when *slot* is not in use, this adheres the specified semantics for *advance*. The rest of the correctness follows from that of *inUse*. ■

The *retreat* method moves the marker to the previous slot in use, or to FORE if there is no such slot. It returns *true* if and only if it retreats to an inuse slot. It throws an *AtBoundaryException* when the marker is already at FORE since then there is no place to retreat.

```
public boolean retreat() throws ConcurrentModificationException {
    if (slot == FORE)
        throw new AtBoundaryException("Already before front.");
    checkValidity();
    while (--slot ≥ 0)                       //slot will be at FORE when loop exits
        if (inUse(slot))
            return true;
    return false;
}
```

Correctness Highlights: The correctness argument is like that for *advance*. Observe then when *slot* is at AFT (*table.length*), when it is decremented it will correctly be at slot *table.length - 1*. Also, note that when *slot* is not in use, *prev(slot)* is the updated value for *slot*. ■

To provide the functionality of the Java *Iterator* interface, we provide the *hasNext* method that returns *true* if there is some element after the current marker location. Since the most typical use of *hasNext* is to follow it by either *next*, for efficiency we move the marker to the slot just before the one holding the next element in the iteration order, unless the marker is already at AFT (in which case it is not moved), or the marker is already at the last element in the iteration order (in which case it is moved to slot *m - 1*).

```

public boolean hasNext() throws ConcurrentModificationException {
    checkValidity();
    if (slot == table.length)                                //already at AFT
        return false;
    boolean ans = advance();                                  //true if there is a next item
    slot--;
    return ans;
}

```

Correctness Highlights: Like with *advance*, if the marker is at AFT then there is no next element in the set. Otherwise, *advance* moves the marker to the next element, from which it moves back one slot (possibly leaving it at slot that is not in use). Although the marker may be left at a slot not in use, $AF(loc)$ is unchanged. The remainder of the correctness follows from that of *advance*. ■

The *remove* method removes the element at the marker, and updates the marker to be at the element in the iteration order that preceded the one removed. It throws a *NoSuchElementException* when the marker is logically at FORE or AFT.

```

public void remove() throws ConcurrentModificationException {
    if (!inCollection())
        throw new NoSuchElementException();
    clearSlot(slot);
}

```

Correctness Highlights: SIZE is preserved since removing an element reduces n by 1, and *size* is also decremented by 1. The remainder of the correctness follows from that of *inCollection* and *clearSlot*. Observe that abstraction function adheres to the required semantics, since $prev(slot)$, is by definition, the element that preceded the removed element in the iteration order. ■

21.6 Performance Analysis

The asymptotic time complexities of the DirectAddressing public methods are given in Table 21.2, and the asymptotic time complexities for all of the public methods of the Marker class are given in Table 21.3.

The complexity of $e.hashCode$ depends upon the length of e in its internal representation. If the size of the internal representation of all elements is a constant, then *hashCode* can be assumed to take constant time. Throughout the remainder of this section we make the assumption that the *hashCode* method for the elements stored in the set takes constant time. For any application where this assumption is not valid, then the DigitizedOrderedCollection ADT should be considered since it can sometimes determine that an element e is not in a set by examining only a small portion of e . Throughout the remainder of this discussion, we also assume that $n \geq m/c$ for some constant c where $m = |U|$. If n does not satisfy this condition, then a different data structure should be considered. Observe that the fraction of slots in use is n/m .

method	worst-case time complexity
<i>ensureCapacity(x)</i> <i>trimToSize()</i>	— —
<i>add(o)</i> <i>contains(o)</i> <i>getEquivalentElement(o)</i> <i>getLocator(o)</i> <i>remove(o)</i>	$O(1)$ $O(1)$ $O(1)$ $O(1)$ $O(1)$
<i>addAll(c)</i> <i>retainAll(c)</i>	$O(c)$ $O(c)$
constructor <i>accept(v)</i> <i>clear()</i> <i>toArray()</i> <i>toString()</i>	$O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$

Table 21.2 Summary of the asymptotic time complexities for the Set ADT when using direct addressing. Recall that for direct addressing $m = |\mathcal{U}|$. Thus *ensureCapacity* and *trimToSize* are not meaningful since the capacity is required to be m . We assume that $n \geq m/4$, which implies that $m = O(n)$.

method	expected time complexity	worst-case time complexity
<i>advance</i> <i>get()</i> <i>hasNext()</i> <i>inCollection()</i> <i>next()</i> <i>remove()</i> <i>retreat()</i>	$O(1)$ $O(1)$ $O(1)$ $O(1)$ $O(1)$ $O(1)$ $O(1)$	$O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$

Table 21.3 Summary of the expected time complexities for the public locator methods when using direct addressing. Recall that $m = |\mathcal{U}|$, and we assume that $n \geq m/c$ for some constant c .

The internal *slotInUse* methods take worst-case constant time. Also, the internal *locate* method also takes constant time since it just makes a call to *hashCode* and a call to *slotInUse*. Thus, *getLocator* takes constant time since it calls *locate* and the marker constructor.

The methods *add*, *contains*, *get*, *getEquivalentElement*, and *remove* take worst-case constant time since they require constant-time in addition to the time of *locate*, which takes worst-case constant time as discussed above. Similarly, *addAll(c)* and *retainAll(c)* take constant time per element in *c*.

The *accept*, *clear*, *toArray*, and *toString* methods iterate through each slot of table, so each takes $O(n)$ time. Likewise, the constructor takes $O(m) = O(n)$ time since constant time is required to initialize each slot to *EMPTY*.

We now analyze the time complexity of the marker methods. Many of the methods use *inCollection* to both check if the marker is logically at an element in the collection, and also to physically move it to its predecessor in the iteration order. Observe that if one assumes that the elements placed in the set are randomly selected from $\mathcal{U}$, then the expected number of slots required to reach the next element is $m/n = O(1)$. However, in the worst-case it might be necessary to traverse n slots before finding the next one in use. Thus, *get*, *inCollection* and *remove* take expected constant time, but in the worst-case can take linear time. Likewise, *advance*, *hasNext*, *next*, and *retreat* have expected $O(1)$ time complexity, and worst case $O(n)$ time complexity since their computation is dominated by finding the next slot in use.

21.7 Quick Method Reference

DirectAddressing Public Methods

- p. 294 *DirectAddressing*(int capacity)
- p. 98 void **accept**(Visitor<? super E> v)
- p. 296 void **add**(E element)
- p. 100 void **addAll**(Collection<? extends E> c)
- p. 100 void **clear**()
- p. 97 boolean **contains**(E value)
- p. 99 void **ensureCapacity**(int capacity)
- p. 96 int **getCapacity**()
- p. 96 Comparator<? super E> **getComparator**()
- p. 98 E **getEquivalentElement** (E target)
- p. 298 Locator<E> **getLocator**(E value)
- p. 96 int **getSize**()
- p. 96 boolean **isEmpty**()
- p. 297 Locator<E> **iterator**()
- p. 297 boolean **remove**(E element)
- p. 100 void **retainAll**(Collection<E> c)
- p. 97 Object[] **toArray**()
- p. 97 E[] **toArray**(E[] array)
- p. 98 String **toString**()
- p. 99 void **trimToSize**()

DirectAddressing Internal Methods

- p. 294 *DirectAddressing*(int capacity, Comparator<? super E> comp)
- p. 293 *DirectAddressing*(int capacity, double load, Comparator<? super E> equivalenceTester, Hasher<? super E> hasher)
- p. 297 boolean **clearSlot**(int slot)
- p. 97 int **compare**(E e1, E e2)
- p. 97 boolean **equivalent**(E e1, E e2)

- p. 295** boolean **inUse**(int slot)
- p. 295** int **locate**(E element)
- p. 99 void **traverseForVisitor**(Visitor<? super E> v)
- p. 98 void **writeElements**(StringBuilder s)

DirectAddressing.Marker Public Methods

- p. 300** boolean **advance**()
- p. 299** E **get**()
- p. 300** boolean **hasNext**()
- p. 101 void **ignoreConcurrentModifications**(boolean ignore)
- p. 101 void **ignorePriorConcurrentModifications**()
- p. 299** boolean **inCollection**()
- p. 101 E **next**()
- p. 301** void **remove**()
- p. 300** boolean **retreat**()

DirectAddressing.Marker Internal Methods

- p. 299** *Marker*(int slot)
- p. 101 void **checkValidity**()
- p. 101 void **updateVersion**()

Chapter 22

Open Addressing Data Structure

package collection.set

AbstractCollection<E> implements Collection<E>

└ DirectAddressing<E> implements Set<E>

└ OpenAddressing<E> implements Set<E>

Uses: Java primitive array

Used By: OpenAddressingMapping (Section 49.7.2)

Strengths: Open addressing is applicable even when only an extremely small fraction of $\mathcal{U}$ will be held in the collection and still guarantees expected constant time performance for all primary methods. If space utilization is of primary importance and less than half of $\mathcal{U}$ is expected to be held in the set then open addressing is an excellent choice, especially if removing elements from the set is relatively rare. Also, both *remove* and *retainAll* are critical mutators only if they cause the hash table to resize, which can be avoided by an appropriate call to *ensureCapacity*.

Weaknesses: For *open addressing* it is crucial that the load factor remains below 1, since otherwise there would not be enough slots in the hash table for all elements. Also, elements that are removed from the set can increase the cost of locating other elements. Like separate chaining, locating an element with open addressing has expected constant cost. With very small probability, it could take linear time to locate a given element.

Critical Mutators: *add* (when it causes resizing), *addAll* (when it causes resizing), *ensureCapacity* (when it causes resizing), *remove* (when it causes resizing), *retainAll* (when it causes resizing), *trimToSize*

Competing Data Structures: Since open addressing is more space efficient than separate chaining, it is very good even when n reaches $|\mathcal{U}|/2$. However, if $n \geq |\mathcal{U}|/2$ then direct addressing (Chapter 21) should be used since the space usage will be less than that of any other approach and it would also be the most time efficient. When space usage is not the primary concern, or when removing elements is a frequent operation, separate chaining (Chapter 23) should be considered.

22.1 Internal Representation

We now present the internal representation for open addressing. When $n < |\mathcal{U}|/2$ it is not space efficient to assign a unique slot in *table* for each object in $\mathcal{U}$. Let $m = \text{table.length}$. If all elements in the set are to be kept in the table, we still require a mechanism for determining a slot for each

element added to the set. This task is accomplished by defining a **hash function** to map each element $x \in \mathcal{U}$, to an integer in $\{0, 1, \dots, m - 1\}$. Specifically, $\text{hash}(\text{hasher.getHashCode}(x))$ returns an integer in between 0 and $m - 1$ (inclusive). Since the hash function is mapping from a very large domain (of size $|\mathcal{U}|$) to a much smaller domain (of size m), collisions, in which two non-equivalent elements hash to the same slot, are unavoidable. The difference between separate chaining and open addressing is in how collisions are resolved. In open addressing, when an element to be inserted hashes to an occupied slot, the collision is resolved by finding a different (empty) slot in which to place the new element. (this is in contrast to separate chaining which creates a chain for each slot to hold all the elements that hash there.)

Throughout the remainder of this chapter we will view the hash table as a circular array where slot 0 is viewed as immediately following slot $m-1$. For each element $e \in \mathcal{U}$, we define a **probe sequence** Π_e , which is a permutation of the slots of the hash table that defines the order in which the slots are considered for element e . More specifically, $\Pi_e = \langle \Pi_e(0), \Pi_e(1), \dots, \Pi_e(m-1) \rangle$. To search for an element, the slots are considered in the order given by the probe sequence until either the element is found, or an empty slot is reached. A new element, is always inserted in the first empty slot reached in the probe sequence. By definition of a permutation, the probe sequence includes each of the m slots exactly once, ensuring that unless all m slots are full, the search for an empty slot will terminate successfully.

We let $\Pi_e(0) = \text{hash}(\text{hasher.getHashCode}(e))$. Since a secondary hash function will be used in defining the probe sequence, hash is often called the **primary hash function**. The simplest method to define the probe sequence is **linear probing**, where for $i = \{1, \dots, m-1\}$,

$$\Pi_e(i) = (\Pi_e(i-1) + 1) \bmod m.$$

That is, each step in the probe sequence moves to the next slot. While linear probing is very simple, it has the drawback that if several elements map to the same slot, then a block of used slots is formed in the hash table. It then becomes more likely that the next element hashes to a slot in this block that causes the block to grow, and so on. This effect is called **primary clustering**.

To avoid primary clustering, one solution is **quadratic probing** that for $i = \{1, \dots, m-1\}$, defines

$$\Pi_e(i) = c_1 \cdot \Pi_e(i-1) + c_2 \cdot (\Pi_e(i-1))^2 \bmod m.$$

This method is superior to linear probing, but it is still the case that if two elements hash to the same initial slot, they follow the same probe sequence. This leads to **secondary clustering**.

The standard solution to avoid secondary clustering is to vary the sequence of slots considered based on the element being inserted. Then, when there is a collision, which already occurs with a low probability, the probability that the same sequence is followed is extremely small. This is accomplished by defining a secondary hash function, **stepHash** that computes the amount to move forward in the hash table whenever the slot being considered is in use. This hash function is often called the **secondary hash function**. More specifically, for $i = \{1, \dots, m-1\}$,

$$\Pi_e(i) = (\Pi_e(i-1) + \text{stepHash}(e.\text{hashCode}())) \bmod m.$$

If stepHash is not designed carefully, the probe sequence might not be a permutation of $\{0, 1, \dots, m-1\}$. For example, suppose that $m = 8$ and $\text{stepHash} = 4$. Regardless of the value of hash , the probe sequence moves back and forth between two elements. For example if the primary hash function is 3 then the probe sequence is $\langle 3, 7, 3, 7, \dots \rangle$. If both of these slots are full then the search for an empty slot will never end.

When stepHash and hash are **relatively prime** (meaning that they have no factor in common), then it is guaranteed that the probe sequence is a permutation of $0, \dots, m-1$. A simple way to ensure this holds is to make m a power of two, and select stepHash to always be odd.

There is one final issue that must be addressed. How will deletions be performed? In order for the search for an element to be efficient, it is crucial that the search can safely terminate at the first empty slot found during the probe sequence. Consider the situation in which element e_1 is at slot 3. Suppose an element e_2 (with a hash value of 3 and a step hash value of 4) is inserted, so e_2 is placed at slot 7. Now suppose e_1 is removed. The most natural approach, is to set $table[3]$ to *EMPTY*. But then the search for e_2 would fail if it stopped at the first empty slot. The standard solution is to signify that an element has been removed from slot s by setting $table[s]$ to the singleton *DELETED*. Recall that we call such a slot a **deleted slot**. Now when a search for an element reaches an empty slot, it is guaranteed that no equivalent element is in the set. However, this solution creates another problem: the hash table gets cluttered with these deleted slots, and even worse, the time to locate an element is affected by them. Unfortunately, we know of no satisfactory solution except to re-use a deleted slot when a newly inserted element's probe sequence contains a deleted slot before an empty slot. Furthermore, deleted slots are counted in computing the load factor, and all deleted slots are eliminated when the hash table is resized. Throughout this chapter we use d to denote the number of deleted slots.

Instance Variables and Constants: Variables *size*, *DEFAULT_CAPACITY*, *FORE*, *version*, and *comp* are inherited from *AbstractCollection*. As with direct addressing, since iteration proceeds by going from slot 0 to $table.size - 1$, the locator implementation is simplified by treating slot -1 as *FORE* and slot $table.length$ as *AFT*.

The application program is given the flexibility to specify the target load which is stored in *targetLoad*. The constant *DEFAULT_LOAD* is the recommended load factor of 0.5. A larger target load factor will decrease space usage at the cost of increasing the expected time complexity of the primary methods. A smaller target load factor will decrease the expected time complexity of the primary methods at the cost of increasing the space usage. The desired load, stored in *targetLoad*, is required to be strictly less than 1.

```
public static final double DEFAULT_LOAD = 0.5;           //default value for target load
double targetLoad;                                     //the desired value for (size+d)/table.length
```

The instance variable d maintains the number of deleted slots, which is needed to compute the current load factor.

```
int d;                                                  //number of deleted slots
```

Similar to our dynamic array implementation (Chapter 13), the remaining constants and instance variables are used to control the automatic resizing of the hash table. While the user provides a desired load factor, the actual load factor, $\alpha = (n+d)/table.size$, must vary since the hash table is only occasionally resized, and its size must be a power of 2. The constants *MIN_RATIO* and *MAX_RATIO* specify how much the actual load can vary from the target load. The variable *minCapacity* holds minimum capacity that should be preserved (excluding a call to *trimToSize*), as specified by the user via the constructor or *ensureCapacity*. The variable *lowWaterMark* gives the value for *size* when the load is at its minimum allowed value, and *highWaterMark* gives the number of slots in use ($size + d$) when load is at its maximum allowed value. The choice for these thresholds is discussed in more depth on page 314.

```
int minCapacity;                                       //min capacity based on ensureCapacity
int lowWaterMark;                                     //min value for size (unless smaller than minCapacity)
int highWaterMark;                                    //max value for size
```

Populated Example: Figure 22.1 shows an open addressing *table* with size $m = 8$ for a set $\{A, B, E, F, H, J\}$, inserted in that order.

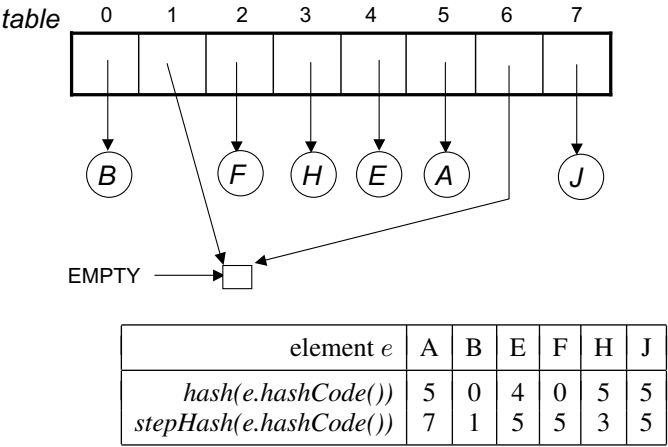


Figure 22.1
A populated example for a set containing the $\{A, B, E, F, H, J\}$, inserted in that order, for a hash table of size $m = 8$ for the given hash values.

Abstraction Function: The abstraction function is the same as that for direct addressing (Chapter 21). Let OA be an open addressing instance. The abstraction function

$$AF(OA) = \{v \in \mathcal{U} \mid \exists i \text{ for which } table[i] = v\}.$$

Terminology: We use the same terminology as for direct addressing, in addition to the following definitions.

- We use $\Pi_e = \langle s_0, s_1, \dots, s_{m-1} \rangle$ to denote the probe sequence defined for element e . That is,

$$s_i = (hash(hashCode(e)) + i \cdot stepHash(hashCode(e))) \bmod m.$$

It is required that for all e , Π_e is a permutation of $\{0, 1, \dots, m - 1\}$.

- We define $stop(e)$ to be the first empty slot in $\Pi(e)$.

22.2 Representation Properties

To simplify our discussion of correctness, we introduce the following representation properties in addition to SIZE inherited from AbstractCollection. At all times the instance variable *load* holds the user supplied value to load (through the constructor or *setLoad*). If no user value has been supplied then *load* holds the default value 0.5. Similarly, *minCapacity* holds the most recent parameter value for *ensureCapacity* (or the initial capacity if there have been no calls to *ensureCapacity*). A call to *trimToSize* is understood to contain an implicit call to *ensureCapacity(0)*. Since the hash table size is rounded up to the nearest power of 2, the hash table size could be up to twice that needed to achieve the desired load.

MEMBERSHIP: If element e is not in the set, then no equivalent element is held in the hash table. More formally, if e is not in the set, for all $0 \leq i \leq m - 1$, $table[s_i] \not\cong e$.

PLACEMENT: If element e is in the set, then it can be found at one of the slots in the probe sequence that precede the first empty slot. More formally, if element e is in the set, for some $0 \leq i \leq stop(e) - 1$, $table[s_i] \cong e$.

MARKEDASDELETED: The instance variable d holds the number of slots refer to the singleton *DELETED*. That is, for $D = \{i \mid table[i] = DELETED\}$, $d = |D|$.

THRESHOLD The *lowWaterMark* is the minimum number of elements for *table* to maintain that expected search time is at most twice as high as it would be at the target load, and the *highWaterMark* is the maximum number of elements for *table* to maintain that the expected search time is no less than half as much as it would be at the target load. More formally,

$$lowWaterMark = \lceil \max\{minCapacity \cdot targetLoad, \lceil table.length \cdot targetLoad / 2 \rceil\} \rceil, \text{ and}$$

$$highWaterMark = \lfloor table.length \cdot (1 + targetLoad) / 2 \rfloor.$$

PROPORTIONALITY The size of the set is between *lowWaterMark* and *highWaterMark* (inclusive), unless an application program call to *ensureCapacity* has forced the number of elements held to go below the *lowWaterMark*. That is, $size \leq highWaterMark$, and either $size \geq lowWaterMark$ or $array.length = minCapacity$.

MINCAPACITY: The capacity of *table* is at least that specified by the application program using *ensureCapacity* and *targetLoad*. More formally,

$$table.length \geq minCapacity / targetLoad.$$

PLACEMENT is crucial to the time complexity analysis for open addressing. This property guarantees that only the slots in the probe sequence up until the first empty slot need be considered to determine if an equivalent element e is held in the set.

22.3 Default Open Addressing Hasher Class

The following open addressing hasher uses slot 0 for the *null* element, and the element's hash code for all other elements.

```
public static class DefaultOpenAddressingHasher<E> implements Hasher<E> {

    public int getHashCode(E element) {
        return (element == null) ? 0 : element.hashCode();
    }

    public int getTableSize(int capacity, double load) {
        return (int) Math.pow(2, Math.ceil(Math.log(capacity/load)/Math.log(2)));
    }
}
```


22.4 Open Addressing Methods

In this section we present the internal and public methods for the `OpenAddressing` class.

22.4.1 Constructors

The most general constructor takes four arguments: *capacity*, the desired capacity, *load*, the target load, *equivalenceTester*, the comparator that defines equivalence of objects, and *hasher*, the user-supplied hash code computation. It creates a hash table with the specified capacity (plus one to handle *null* values), and with the given comparator and hasher. It throws an *IllegalArgumentException* when *capacity* < 0 or *load* ≥ 1.0. Since the elements held in the mapping could be *null*, *EMPTY* is used instead of *null* for initialization.

```
public OpenAddressing(int capacity, double load,
                     Comparator<? super E> equivalenceTester, Hasher<E> hasher){
    super(capacity, load, equivalenceTester, hasher);
    if (load ≥ 1.0)
        throw new IllegalArgumentException("Load must be < 1");
    this.targetLoad = load;
    minCapacity = capacity;
    d = 0;
    updateResizingBounds();
}
```

Correctness Highlights: Since the desired value for $load = (n + d)/m = n/m$, the capacity of the table should be allocated to be the smallest power of two at least $capacity/load$. Thus `MINCAPACITY` is satisfied. Since the original capacity for the hash table is the smallest power of two at least as large as $\lceil minCapacity/targetLoad \rceil$, `PROPORTIONALITY` is satisfied. Setting $d = 0$, satisfies `MARKEDASDELETED`. `PLACEMENT` is trivially satisfied since no elements are in the set, and `SIZE` is satisfied by the direct addressing constructor. Finally *updateResizingBounds* satisfies `THRESHOLD`. ■

Several additional convenience constructors are provided to replace some parameters by the default values. Their correctness follows from that of the above constructor. The constructor that takes three arguments: *capacity*, the desired capacity, *load*, the target load, and *equivalenceTester*, the comparator that defines equivalence of objects, creates a hash table with the specified capacity (plus one to handle *null* values), and with the given comparator and default open addressing hasher. It throws an *IllegalArgumentException* when *capacity* < 0 or *load* ≥ 1.0.

```
public OpenAddressing(int capacity, double load,
                     Comparator<? super E> equivalenceTester) {
    this(capacity, load, equivalenceTester, new DefaultOpenAddressingHasher<E>());
}
```

The constructor that takes no arguments creates a new set with a default initial capacity, load, equivalence tester, and hasher.

```
public OpenAddressing(){
    this(DEFAULT.CAPACITY, DEFAULT.LOAD, DEFAULT.EQUIVALENCE.TESTER);
}
```

The constructor that takes the single argument, *equivalenceTester*, the comparator defining equivalence, creates a new set with a default initial capacity, load, and hasher.

```

public OpenAddressing(Comparator<? super E> equivalenceTester) {
    this(DEFAULT_CAPACITY, DEFAULT_LOAD, equivalenceTester);
}

```

The constructor with the single argument *capacity*, the desired capacity for the set, creates a new set with the given capacity, and the default load, equivalence tester, and hasher.

```

public OpenAddressing(int capacity) {
    this(capacity, DEFAULT_LOAD, DEFAULT_EQUIVALENCE_TESTER);
}

```

The constructor that takes the two arguments *capacity*, the desired capacity, and *equivalenceTester*, the comparator defining equivalence, creates a new set with the provided capacity and equivalence tester, using the default load and hasher.

```

public OpenAddressing(int capacity, Comparator<? super E> equivalenceTester) {
    this(capacity, DEFAULT_LOAD, equivalenceTester);
}

```

The constructor that takes the two arguments *capacity*, the desired capacity, and *load*, the desired load factor, creates a new set with the provided capacity and load, using the default equivalence tester and hasher. It throws an *IllegalArgumentException* when *capacity* < 0 or *load* ≥ 1.0.

```

public OpenAddressing(int capacity, double load){
    this(capacity, load, DEFAULT_EQUIVALENCE_TESTER);
}

```

22.4.2 Trivial Accessors

The *getSize* and *isEmpty* methods are inherited from *AbstractCollection*. The *getCapacity* class returns the capacity of the underlying representation for the current hash table to meet the desired load factor.

```

public int getCapacity() {
    return (int) Math.floor(table.length*targetLoad-d);
}

```

Correctness Highlights: Since the actual load is $\alpha = (n + d)/m$, solving for n yields that the capacity is $m \cdot \alpha - d$, which is the value returned since *table.length* = m and *load* = α . ■

22.4.3 Representation Accessors

The internal *inUse* method takes *slot*, the desired slot, and returns *true* if and only if *slot* is in use. It requires that *slot* is a valid slot number (i.e., $0 \leq \text{slot} < \text{table.length}$).

```

boolean inUse(int slot){
    return table[slot] != EMPTY && table[slot] != DELETED;
}

```

The *clearSlot* method takes *slot*, a slot to be marked as deleted. It requires that the slot is in use.

```

boolean clearSlot(int slot) {
    if (inUse(slot)) {
        table[slot] = DELETED;
        d++;
        size--;
        return true;
    } else
        return false;
}

```

//preserves Placement
//preserves MarkedAsDeleted
//preserves Size

Correctness Highlights: By setting the slot to *DELETED*, *PLACEMENT* is preserved, and by incrementing *d*, *MARKEDASDELETED* is preserved. Note that because the sum of *d* and *size* is unaffected, it is not necessary to consider resizing the table. ■

22.4.4 Selecting a Hash Function

In selecting a hash function, the critical property required for good performance is that each object in $\mathcal{U}$ has probability $1/m$ of hashing to each slot of the hash table. When this is the case then the expected number of elements from the collection that hash to each slot is n/m . There is a trade-off between the cost of computing the hash function and the need to ensure that each object in $\mathcal{U}$ is equally likely to hash to each slot. A very efficient hash function is $\text{hash}(i) = i \bmod m$. When m is a power of 2 this can be computed as a shift. This method for computing a hash function is called the **division method**.

When the *hashCode* function is designed appropriately and the elements inserted into the collection are *randomly* selected from $\mathcal{U}$ the division method yields the desired probabilistic property. However, typically the elements inserted into the collection are not randomly selected, so a more sophisticated hash function is needed to ensure that each element is equally likely to hash to each of the m slots. For example, consider when the bar code for a product is used to define equivalence, and suppose that the least significant digits encode the manufacturer. Using $i \bmod m$ as a hash function would lead to many hash values being the same. While this is a contrived example, it is common for real applications to have some patterns in the elements.

A very good choice for hash functions is the **multiplication method**, which is designed so that a random hash code is equally likely to hash to each of the m possible values. The intuition behind the multiplication method is that it computes $x * A - \lfloor k * A \rfloor$ for a carefully selected constant A and then returns only the most significant digits of the resulting fraction (modulo m). Based on extensive experimentation, Knuth [97] suggests using the inverse of the golden ratio, $A = (\sqrt{5} - 1)/2$.

```

public static double A = (Math.sqrt(5.0)-1)/2;

```

//multiplier for hash function

The *hash* method takes x , an arbitrary integer. It returns an integer between 0 and $m - 1$.

```

protected int hash(int x) {
    double frac = (x * A) - (int) (x * A);
    int hashValue = (int) (table.length * frac);
    if (hashValue < 0)
        hashValue += table.length;
    return hashValue;
}

```

//fractional part of x*A
//multiply by m
//if this is negative add m to get
//hashValue mod m

Correctness Highlights: The intuition as to why the multiplication method distributes the elements evenly, even when there are patterns in the data, is that A is selected to be a real number less than 1 that does not exhibit any patterns. Consider a wheel that is divided into m equal-sized wedges and spun. Because of the choice of A , the product of $x * A$ simulates spinning the wheel where it ends in a random wedge. (The non-fractional portion of $x * A$ corresponds to how many times the wheel went around which is not random, but also is not used since patterns in the data would affect its value.) Thus $frac$ corresponds to the final spinner position. Multiplying m by $frac$ computes which of the pieces the spinner is in. Finally, since x could be a negative value between -1 and $-m$ could be obtained, which is converted to a value between 0 and $m - 1$ by adding m . This value is then returned. ■

The secondary hash function used for the step size must return a value that is relatively prime to the `table.length`. Our implementation ensures that they are relatively prime by making m a power of two, and ensuring that step hash is odd. Also, to remove the dependence on any patterns in the low order bits of `hashKey`, we use the division method with value $m/2 - 1$.

```
int stepHash(int hashKey) {
    int s = (hashKey % (table.length/2 - 1));
    if (s < 0)
        s += (table.length/2 - 1);
    return 2*s + 1;
}
```

Correctness Highlights: The definition of s ensures that $|s| < m/2 - 1$. By adding $m/2 - 1$ to s when it is negative, we have the property that s is an integer between 0 and $m/2 - 2$. Furthermore, if `hashKey` is uniformly distributed, then s is uniformly distributed among these $m/2 - 1$ integers. Finally since the value returned is $2s + 1$, we ensure that the value for the step hash function is an odd number between 1 and $m - 3$ making it relatively prime with any power of two. ■

22.4.5 Algorithmic Accessors

All algorithmic accessors are inherited from `DirectAddressing`, except for `locate` that takes `target`, the element to search for, and returns the slot referencing an equivalent element to `target`, or the insert position if no equivalent element is in the set.

```
protected int locate(E target) {
    int hashCode = hasher.getHashCode(target);
    int index = hash(hashCode); //first slot in probe sequence
    int step = stepHash(hashCode); //step size between probes
    int insertPosition = FORE;
    while (table[index] != EMPTY) { //continue until an empty slot found
        if (table[index] != DELETED) {
            if (equivalent((E) table[index], target))
                return index;
        } else if (insertPosition == FORE) {
```

```

        insertPosition = index;
    }
    index = (index + step) % table.length;           //move forward in probe sequence
}
if (insertPosition == FORE)
    insertPosition = index;
return insertPosition;
}

```

Correctness Highlights: If the target is found, clearly its index is correctly returned. By PLACEMENT and the correctness of the equivalence tester, we know that if the loop terminates without finding the element then there is no equivalent element in the set. In that case, the first slot encountered during the search that was not in use is returned as the insert position for that element. ■

22.4.6 Representation Mutators

We first define method *updateResizingBounds* that sets the values for *lowWaterMark* and *highWaterMark*. Since these values determine when resizing is required, this method is called upon initialization, each time the table is resized, and whenever the load factor changes. As discussed in the analysis section, the expected number of probes in an unsuccessful search is $1/(1 - \alpha)$. When the current load is less than half of the target load, the table is underutilized and should be resized. As an example, if $\alpha = 1/2$, then ideally half of the slots in the table should be in use. For this load, the low water mark is reached when the table is $1/4$ utilized.

Recall that for open addressing the load must be less than 1. Thus, the high water mark cannot be set to the size where the table utilization is twice the desired value. Instead, the high water mark is set at the point where the hash table utilization is midway between the desired value and 1. This load corresponds to the point when the expected number of non-empty slots accessed during an unsuccessful search is twice the desired value. As an example, if $\alpha = 1/2$, the high water mark is reached when the table is $3/4$ utilized.

```

void updateResizingBounds() {
    lowWaterMark = (int) Math.max(minCapacity*targetLoad,
                                   Math.ceil(table.length*targetLoad/2));
    highWaterMark = (int) (table.length*(1+targetLoad)/2);
}

```

The *resizeTableAsNeeded* method resizes the table to preserve PROPORTIONALITY.

```

void resizeTableAsNeeded() {
    if (size < lowWaterMark || size + d > highWaterMark)
        resizeTable(Math.max(size, minCapacity));
}

```

The *growTableAsNeeded* method increases the table size when the number of elements held in the set plus the number of slots marked as deleted has reached the high water mark.

```

void growTableAsNeeded() {
    if (size + d > highWaterMark)
        resizeTable(Math.max(size, minCapacity));
}

```

The *shrinkTableAsNeeded* method reduces the table size when the number of elements held in the set drops to the lower water mark.

```
void shrinkTableAsNeeded() {
    if (size < lowWaterMark)
        resizeTable(Math.max(size, minCapacity));
}
```

The internal *resizeTable* method takes *desiredCapacity*, the number of elements that can be placed in this set while maintaining the load factor of *targetLoad*. Since the hash function depends on the hash table size, when resizing the hash table, all elements must be moved, which invalidates all active markers for iteration. This method guarantees that *m* is a power of 2 by rounding up the ideal table size to the next power of 2. A beneficial side effect of resizing is that the table is cleaned of all deleted slots, and *d* is reset to 0.

```
void resizeTable(int desiredCapacity){
    OpenAddressing<E> newTable = new OpenAddressing<E>(desiredCapacity,
                                                         targetLoad, comp);

    for (E e: this)                                     //insert all elements
        newTable.add(e);                               //into the new table
    this.table = newTable.table;
    d = 0;
    version.increment();                               //invalidate all active markers
    updateResizingBounds();
}
```

Correctness Highlights: Since $\alpha = (n + d)/m$, for a given load and current *size*, the capacity for table that will satisfy the given load is n/α since *d* will be 0 after resizing. This value is rounded up to the next power of two and used as the new table capacity. By PLACEMENT only the elements in the set are added to *newTable*, and the other slots will be empty. By the correctness of the *add* method, SIZE and PLACEMENT are preserved. The version number is incremented since all active markers must be invalidated. ■

The *setLoad* method takes *load*, the desired load, and resizes the table so that it has the desired load factor.

```
public void setTargetLoad(double load){
    this.targetLoad = load;
    updateResizingBounds();
    resizeTableAsNeeded();
}
```

Correctness Highlights: By the correctness of *resizeTableAsNeeded*, PROPORTIONALITY is preserved. The rest of the correctness follows from that of *resizeTable*. ■

The *ensureCapacity* method takes *desiredCapacity*, the desired capacity, and resizes the table if necessary so that it has the desired capacity.

```
public void ensureCapacity(int desiredCapacity){
    minCapacity = desiredCapacity;
    updateResizingBounds();
    growTableAsNeeded();
}
```

Correctness Highlights: The first line preserves MINCAPACITY. THRESHOLD is preserved by *updateResizingBounds*. The rest of the correctness follows from that of *growTableAsNeeded*. ■

The *trimToSize* method adjusts the size of the hash table so that the desired load factor is met as long as this does not increase the hash table size.

```
public void trimToSize(){
    minCapacity = Math.min(minCapacity, size);
    updateResizingBounds();
    shrinkTableAsNeeded();
}
```

Correctness Highlights: The first line preserves MINCAPACITY. THRESHOLD is preserved by *updateResizingBounds*. The rest of the correctness follows from that of *shrinkTableAsNeeded*. ■

22.4.7 Content Mutators

The *add* method takes *element*, the element to be inserted into this set. Let *s* be the first empty slot in the probe sequence for *element*. Once it is determined that there is no element equivalent in the set, the first deleted slot that occurred, if any, before *s* is re-used. If no deleted slots were found, then slot *s* is used.

```
public void add(E element) {
    int slot = locate(element);
    if (!inUse(slot)) {
        size++;
        if (table[slot] == DELETED)
            d--;
    }
    table[slot] = element;
    growTableAsNeeded();
}
```

//get insert position
//element needs to be added

//invalidates active markers

Correctness Highlights: By the correctness of *locate*, *slot* is the slot at which the new element should be placed. By PLACEMENT and the correctness of *inUse*, when *slot* is not in use, then no equivalent element is in the set. Since the size of the set will grow by one, incrementing the instance variable *size* preserves SIZE. Also, when *slot* is marked as deleted, decrementing *d* preserves MARKEDASDELETED. PLACEMENT is preserved by placing the new element into *slot*. The rest of the correctness follows from that of *growTableAsNeeded*. ■

The *remove* method takes *element*, the target, and removes an equivalent element (if one exists) from the set. It returns *true* if and only if the element was in the set and hence removed.

```
public boolean remove(E element) {
    if (super.remove(element)){
        resizeTableAsNeeded();
        return true;
    }
```

method	expected time complexity	worst-case time complexity
<i>add(o)</i>	$O(1)$	$O(1)$
<i>contains(o)</i>	$O(1)$	$O(n)$
<i>getEquivalentElement(o)</i>	$O(1)$	$O(n)$
<i>getLocator(o)</i>	$O(1)$	$O(n)$
<i>remove(o)</i>	$O(1)$	$O(n)$
<i>ensureCapacity(x)</i>	$O(n + x)$	$O(n + x)$
<i>addAll(c)</i>	$O(c)$	$O(c)$
<i>retainAll(c)</i>	$O(c)$	$O(c \cdot n)$
constructor	$O(n)$	$O(n)$
<i>accept(v)</i>	$O(n)$	$O(n)$
<i>clear()</i>	$O(n)$	$O(n)$
<i>toArray()</i>	$O(n)$	$O(n)$
<i>toString()</i>	$O(n)$	$O(n)$
<i>trimToSize()</i>	$O(n)$	$O(n)$

Table 22.2 Summary of the asymptotic time complexities for the *Set* public methods when using the open addressing data structure when $m \leq c \cdot n$ for some constant c .

Set

```
    }  
    return false;  
}
```

Correctness Highlights: SIZE and MEMBERSHIP are maintained by the superclass *remove* method. The call to *resizeTableAsNeeded* ensures that THRESHOLD and PROPORTIONALITY are preserved. Finally *true* is correctly returned exactly when the superclass method returns *true*. ■

The *clear* method removes all elements from the collection.

```
public void clear() {  
    super.clear();  
    d = 0; //preserved MarkedAsDeleted  
}
```

Correctness Highlights: Follows from the DirectAddressing *clear* method, and the fact that setting *d* to 0 preserves MARKEDASDELETED. ■

22.5 Performance Analysis

The asymptotic time complexities of all public methods for OpenAddressing are given in Table 22.2. The Locator methods and the analysis of them are exactly the same as for direct addressing as given in Section 21.6. Throughout our analysis, we make the assumption that the *hashCode* takes constant time, and that $m \leq c \cdot n$ for some constant c . Thus $\alpha = n/m$ is a constant.

The time complexities for all Set methods that must locate an element are dominated by the number of probes until the desired element or an empty slot is located. The rest of the computation clearly takes constant time. We assume that $m = \Theta(n)$. If this is not the case, then through *ensureCapacity* the table has been significantly oversized.

Under the standard assumption that the hash function is selected such that each element is equally likely to hash to each of the m slots, the probability that the desired item is in any given slot is α . One can view the probe sequence of trying all m slots in a random order (as defined by *hash* and *stepHash* so that the same sequence is repeated each time). We now provide the intuition behind why the expected number of probes is $1/(1 - \alpha)$. Probe $\Pi(0)$ always occurs. The probability that slot $\Pi(0)$ is occupied is α in which case a second probe must be made. The probability that the first two slots probes are occupied is $n/m \cdot (n - 1)/(m - 1) \approx \alpha^2$. In general, since a small number of probes are made, the probability that the first p slots probed are in use is roughly α^p . Formally, the expected number of probes for an unsuccessful search is $1/(1 - \alpha) = 1 + \alpha + \alpha^2 + \alpha^3 + \dots$. For a successful search, the expected number of probes is $\frac{1}{\alpha} \ln \frac{1}{1 - \alpha}$. A formal derivation for both of these bounds can be found in Knuth [97]. Since $\alpha = (n + d)/m$, with open addressing the expected number of probes in an unsuccessful search is $m/(m - (n + d))$. For a successful search the expected number of probes is roughly half this amount. For any methods that iterate through the entire collection, constant time is spent at each slot, leading to an overall worst-case time complexity of $O(m) = O(n)$.

For open addressing, the space usage is dominated by that of the hash table which has m references. Beyond that, there is just the space for instance variables and constants, and a small amount of space included with each object by the Java run time system.

22.6 Quick Method Reference

OpenAddressing Public Methods

- p. 310 *OpenAddressing()*
- p. 310 *OpenAddressing(Comparator<? super E> equivalenceTester)*
- p. 311 *OpenAddressing(int capacity)*
- p. 311 *OpenAddressing(int capacity, Comparator<? super E> equivalenceTester)*
- p. 311 *OpenAddressing(int capacity, double load)*
- p. 310 *OpenAddressing(int capacity, double load, Comparator<? super E> equivalenceTester)*
- p. 310 *OpenAddressing(int capacity, double load, Comparator<? super E> equivalenceTester, Hasher<E> hasher)*
- p. 98 void **accept**(Visitor<? super E> v)
- p. 296 void **add**(E element)
- p. 100 void **addAll**(Collection<? extends E> c)
- p. 100 void **clear**()
- p. 97 boolean **contains**(E value)
- p. 99 void **ensureCapacity**(int capacity)
- p. 96 int **getCapacity**()
- p. 96 Comparator<? super E> **getComparator**()
- p. 98 E **getEquivalentElement**(E target)
- p. 298 Locator<E> **getLocator**(E value)
- p. 96 int **getSize**()
- p. 96 boolean **isEmpty**()
- p. 297 Locator<E> **iterator**()
- p. 297 boolean **remove**(E element)
- p. 100 void **retainAll**(Collection<E> c)

- p. 315** void **setTargetLoad**(double load)
- p. 97 Object[] **toArray**()
- p. 97 E[] **toArray**(E[] array)
- p. 98 String **toString**()
- p. 99 void **trimToSize**()

OpenAddressing Internal Methods

- p. 297 boolean **clearSlot**(int slot)
- p. 97 int **compare**(E e1, E e2)
- p. 97 boolean **equivalent**(E e1, E e2)
- p. 314** void **growTableAsNeeded**()
- p. 312** int **hash**(int x)
- p. 295 boolean **inUse**(int slot)
- p. 295 int **locate**(E element)
- p. 315** void **resizeTable**(int desiredCapacity)
- p. 314** void **resizeTableAsNeeded**()
- p. 315** void **shrinkTableAsNeeded**()
- p. 313** int **stepHash**(int hashKey)
- p. 99 void **traverseForVisitor**(Visitor<? super E> v)
- p. 314** void **updateResizingBounds**()
- p. 98 void **writeElements**(StringBuilder s)

Chapter 23

Separate Chaining Data Structure

package `collection.set`

`AbstractCollection<E>` implements *Collection<E>*

↳ `SeparateChaining<E>` implements *Set<E>*

Uses: Java primitive array

Used By: `SeparateChainingMapping` (Section 49.7.3)

Strengths: Separate chaining is applicable even when only an extremely small fraction of $\mathcal{U}$ will be held in the collection, and it guarantees expected constant time performance for all primary methods. In addition, separate chaining very naturally and easily can remove elements from the collection, and minimizes the need to resize the underlying array when n is changing significantly. For separate chaining, the only mutations that invalidate the markers are those that remove elements or that resize the table. Thus, calls to *add* or *addAll* that do not cause the table to resize are not critical. An application program can ensure that iteration can always continue with a consistent order in spite of concurrent additions to the set by calling *ensureCapacity* with an appropriate value before the iterator is initialized.

Weaknesses: The disadvantage of separate chaining is that collisions are resolved by creating a singly linked list of all elements that hash to a given slot. Thus, $m + n$ references are required in addition to the n references to the elements, leading to a space usage proportional to at least $m + 2n$ which is higher than the other data structures. For any object-oriented language with polymorphism, the compiler must maintain the type of the object to dispatch the method calls. In any such language, the space usage is really $m + 3n$ since there is a third hidden reference for each list item. Unlike direct addressing that has worst-case constant cost to locate an element in the collection, separate chaining only has expected constant cost. With very small probability, it could take linear time to locate a given element. Also, the *remove* method is a critical mutator even when the table is not resized.

Critical Mutators: *add* (when it causes resizing), *addAll* (when it causes resizing), *clear*, *ensureCapacity*, *remove*, *retainAll*, *trimToSize*

Competing Data Structures: If $n \geq |\mathcal{U}|/4$ then direct addressing (Chapter 21) should be used since the space usage will be less than that of any other approach that provides constant time access. Also, if worst-case constant cost for insertion, deletion, and membership is essential, even at the cost of extra space usage, then direct addressing is best. When space usage is the primary concern, or when the application can benefit from *remove* not being a critical mutator, then open addressing (Chapter 22) is a better choice.

23.1 Internal Representation

We now present the internal representation for separate chaining. As discussed in Chapter 22, when $n < |\mathcal{U}|/2$ it is not space efficient to assign a unique slot in *table* for each object in $\mathcal{U}$. However, if elements are to be kept in the table, we still require a mechanism for determining a slot for each element added to the set. We let $m = \text{table.length}$. As in open addressing, a hash function is used to map any element $x \in \mathcal{U}$, to an integer in $\{0, 1, \dots, m-1\}$. Since the hash function is mapping from a very large domain (of size $|\mathcal{U}|$) to a much smaller domain (of size m), collisions are unavoidable. The difference between separate chaining and open addressing is how collisions are resolved.

In separate chaining, the collisions are resolved by having *table[i]* reference a singly linked list with an entry for each element in the collection that hashes to slot i . We refer to the list for slot i as **chain i** . Another common name given when collisions are resolved in this manner is **hashing with chaining**. While an instance of the `SinglyLinkedList` class could be used for each chain, we instead optimize the space usage for the situation in which each chain only has a few elements. Most notably, we do not use a sentinel head, and perform all insertions to the front of the list removing the need to maintain a reference to the last item in the list.

Instance Variables and Constants: Variables *size*, *DEFAULT_CAPACITY*, *FORE*, *version*, and *comp* are inherited from `AbstractCollection`. As with direct addressing, since iteration proceeds by going from slot 0 to *table.size* - 1, the locator implementation is simplified by treating slot -1 as *FORE* and slot *table.length* as *AFT*. The hash table is an array of references to chain items, where *table[i]* serves as a head pointer for chain i .

```
ChainItem<E>[] table; //the hash table
```

As in open addressing, the application program is given the flexibility to specify the target load which is stored in *targetLoad*. However, for separate chaining, the default load is the recommended value of 1.0. A larger target load factor will decrease space usage at the cost of increasing the expected time complexity of the primary methods. A smaller target load factor will decrease the expected time complexity of the primary methods at the cost of increasing the space usage. The actual load factor, $\alpha = n/\text{table.size}$, is dependent on the hash table size, which should not change often. Unlike open addressing, for separate chaining there is no strict upper bound on the target load, but it is recommended that it is at most 10.

```
public static final double DEFAULT_LOAD = 1.0; //default value for target load
double targetLoad; //the desired value for size/table.length
```

The remaining constants, and instance variables are used to control automatic resizing of the hash table. There are many similarities between this aspect of separate chaining and our dynamic array implementation (Chapter 13). An important distinction is that for separate chaining, the capacity of the collection is defined as the number of elements that can be held while maintaining the desired load factor. That is, the capacity of *table* is *load* * *table.length*. The remainder of this chapter assumes the reader is familiar with the dynamic array implementation provided in Chapter 13.

Although the user provides a desired load factor, the actual load factor ($\alpha = n/m$ where $m = \text{table.size}$) must vary since resizing the hash table is expensive, and should be done infrequently. The constants *MIN_RATIO* and *MAX_RATIO* specify how much the actual load can vary from the target load. In particular, they specify the constraint that

$$\text{targetLoad} \cdot \text{MIN\_RATIO} \leq \frac{n}{m} \leq \text{targetLoad} \cdot \text{MAX\_RATIO}.$$

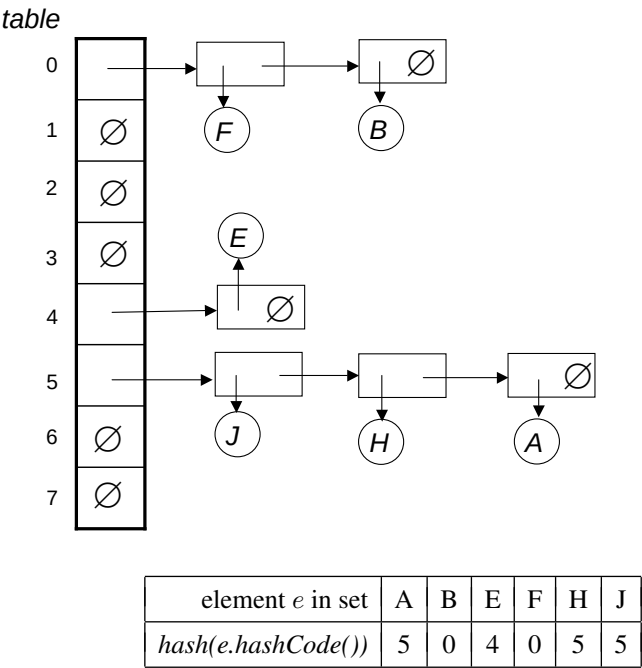


Figure 23.1
A populated example for the set $\{A, B, E, F, H, J\}$ for a hash table of size $m = 8$ with separate chaining using the given hash values.

As done for the dynamic array, *minCapacity* holds the minimum capacity that should be preserved (excluding a call to *trimToSize*), as specified by the user via the constructor or *ensureCapacity*. The variable *lowWaterMark* is the value for *size* when the load is at its minimum allowed value, and *highWaterMark* is the value for *size* when load is at its maximum allowed value. The default values for the *lowWaterMark* and *highWaterMark* are designed to increase the hash table size when the expected number of elements accessed during a search is twice what it would be at the target load. Similarly, the hash table size is decreased when the expected number of elements accessed during a search is half what it would be at the target load. When the hash table is resized, the actual load is made equal to the target load.

```
static final double MIN_RATIO = 0.5;           //keep at or above half the target load
static final double MAX_RATIO = 2.0;           //keep at or below twice the target load
int minCapacity;                               //min capacity to maintain based on ensureCapacity
int lowWaterMark;                              //min value for size (unless smaller than minCapacity)
int highWaterMark;                             //max value for size
Hasher<? super E> hasher;                      //for computing hash codes and the new table size
```

Populated Example: Figure 23.1 shows the internal representation with separate chaining for a set containing $\{A, B, E, F, H, J\}$, inserted in that order, into a hash table of size $m = 8$.

Terminology: We use the following definitions throughout this chapter, in addition to those defined for all collections and sets.

- We define **chain i** to be the chain referenced by *table*[i]. When *table*[i] is *null*, we say that **chain i is empty**.

- We say that chain i **contains** element e if and only if chain i is not empty, and there exists a chain item x in chain i , for which $x.element \cong e$. An empty chain does not contain any elements.

Abstraction Function: Let SC be a `SeparateChaining` instance. The abstraction function

$$AF(SC) = \{e \mid \text{for some valid slot } i, \text{ chain } i \text{ contains } e\}.$$

Optimizations: If it is known that the hash codes for the elements inserted are randomly and uniformly distributed among the integers with no patterns, then some overhead can be removed by using the hash function $hash(i) = i \bmod m$. However, if this assumption fails, such a hash function could lead to a very inefficient implementation. Therefore, it is usually best to use the hash function provided here, which greatly reduces the chance of a larger number of elements hashing to the same slot. If desired, the table capacity can be rounded up to the next power of two so division by m can be performed via shift operations. See the `resizeTable` method on page 315 in open addressing (Chapter 22) to see how this computation can be performed.

In the provided implementation, a chain holding one element is represented as single chain item with a `null` next pointer. Space usage could be reduced if the last item was an element (versus a chain item). The trie data structure (Chapter 41) illustrates how to create a data structure composed of two distinct types of objects. Since it is expected that most chains have a single element, this could reduce space usage by about 30%, but at the cost of including extra conditions in the code which would slightly increase the time complexity. However, if minimizing space usage is important, open addressing should be considered.

Design Notes: In our implementation, each marker encapsulates both a slot number and a reference to the chain item in that slot. When the element in chain item x is removed, there is no efficient mechanism to update the markers that reference x , except that the marker that called `remove` is moved to the predecessor of x . Therefore, `remove` is a critical mutator. To enable iteration to continue properly even when an element is removed, as long as the hash table is not resized, one could implement the same mechanism used in `SinglyLinkedList` (Chapter 15). More specifically, the element in the chain item could be made of type `Object`, and set to a singleton `REMOVED`, and then the `next` reference of the chain item could be used to create a redirect chain.

23.2 Representation Properties

To simplify our discussion of correctness, we introduce the following representation properties in addition to `SIZE` inherited from `AbstractCollection`. `PLACEMENT` specifies where an element must be placed within the hash table, and `INUSE` specifies how a chain item is marked as deleted.

We also introduce `THRESHOLD`, `PROPORTIONALITY`, and `MINCAPACITY` that relate to the automatic resizing. These properties, and their use, are exactly like the corresponding properties in the dynamic array (Chapter 13). The only difference is in the way that the high water mark and low water mark are computed. The target load is exactly maintained when $size = table.length * targetLoad$. By multiplying this product by `MIN_RATIO` (for `getLowWaterMark`) and by `MAX_RATIO` (for `getHighWaterMark`), we compute the values defined in `THRESHOLD`.

PLACEMENT: An element $e \in \mathcal{U}$ is in this set if and only if $table[e.hashCode()]$ contains e . Thus, if $table[e.hashCode()]$ does not contain e , then there is no element equivalent to e in this set.

INUSE: For any chain item x , $x.next = x$ if and only if it holds an element that has been removed from the set.

THRESHOLD The *lowWaterMark* is the minimum number of elements for *table* to maintain that the actual load is at least *MIN_RATIO* times the desired load. Likewise, the *highWaterMark* is the maximum number of elements for *table* to maintain that the actual load is at most *MAX_RATIO* times the desired load. More formally,

$$lowWaterMark = \lceil table.length \cdot targetLoad \cdot MIN\_RATIO \rceil, \text{ and}$$

$$highWaterMark = \lfloor table.length \cdot targetLoad \cdot MAX\_RATIO \rfloor.$$

PROPORTIONALITY The size of the set is between *lowWaterMark* and *highWaterMark* (inclusive), unless an application program call to *ensureCapacity* has forced the number of elements held to go below the *lowWaterMark*. That is, $size \leq highWaterMark$, and either $size \geq lowWaterMark$ or $array.length = minCapacity$.

MINCAPACITY: The capacity of *table* is at least that specified by the application program using *ensureCapacity* and *targetLoad*. More formally,

$$table.length \geq minCapacity / targetLoad.$$

Set

23.3 Chain Item Inner Class

ChainItem<E>

The chains are constructed of chain items, which each hold a reference to an element in the collection, and reference to the next chain item (or *null* for the last item).

E element;
ChainItem<E> next;

//element in the collection
//pointer to next item in the chain

The *ChainItem* constructor takes *element*, the element to store in the chain item, and *next*, a reference to the next item.

```
ChainItem(E element, ChainItem<E> next) {  
    this.element = element;  
    this.next = next;  
}
```

To support the *Marker* class, the *markDeleted* method marks the chain item as deleted by setting the *next* reference to itself.

```
void markDeleted() {  
    next = this;  
}
```

The *isDeleted* method, returns *true* if and only if this chain item has been deleted.

```

boolean isDeleted() {
    return next == this;
}

```

23.4 Default Separate Chaining Hasher Class

The following separate chaining hasher uses slot 0 for the *null* element, and the element's hash code for all other elements.

```

public static class DefaultSeparateChainingHasher<E> implements Hasher<E> {
    public int getHashCode(E element) {
        return (element == null) ? 0 : element.hashCode();
    }
    public int getTableSize(int capacity, double load) {
        return (int) Math.ceil(capacity/load);
    }
}

```

23.5 Separate Chaining Methods

In this section we present the internal and public methods for the `SeparateChaining` class.

23.5.1 Constructors

The most general constructor takes four arguments: *capacity*, the desired capacity, *load*, the target load, *equivalenceTester*, the comparator that defines equivalence of objects, and *hasher*, the user-supplied hash code computation. It creates a hash table with the specified capacity (plus one to handle *null* values), and with the given comparator and hasher. It throws an *IllegalArgumentException* when *capacity* < 0. Since the elements held in the mapping could be *null*, *EMPTY* is used instead of *null* for initialization.

```

public SeparateChaining(int capacity, double load,
    Comparator<? super E> equivalenceTester, Hasher<? super E> hasher){
    super(equivalenceTester);
    minCapacity = capacity;                                //satisfy Capacity
    targetLoad = load;
    this.hasher = hasher;
    size = 0;
    table = (ChainItem<E>[]) new ChainItem[hasher.getTableSize(capacity, load)];
    size = 0;                                              //satisfy Size
    updateResizingBounds();
}

```


Correctness Highlights: The target load is the desired value for n/m . The size of the table should be $capacity/load$ since $capacity$ specifies the value of n that the set should be able to accommodate while maintaining the desired load. Thus, MINCAPACITY is satisfied. SIZE is satisfied since $n = size = 0$. PLACEMENT and INUSE are trivially satisfied since no elements are in the set and all chains are empty. Since the original capacity for the hash table is $\lceil minCapacity/targetLoad \rceil$, PROPORTIONALITY is satisfied. Finally *updateResizingBounds* satisfies THRESHOLD. ■

Several additional convenience constructors are provided to replace some parameters by the default values. Their correctness follows from that of the above constructor. The constructor that takes three arguments: *capacity*, the desired capacity, *load*, the target load, and *equivalenceTester*, the comparator that defines equivalence of objects, creates a hash table with the specified capacity (plus one to handle null values), and with the given comparator and default separate chaining hasher. It throws an *IllegalArgumentException* when *capacity* < 0.

```
public SeparateChaining(int capacity, double load,
    Comparator<? super E> equivalenceTester) {
    this(capacity, load, equivalenceTester, new DefaultSeparateChainingHasher<E>());
}
```

The constructor that takes no arguments creates a new set with a default initial capacity, load, equivalence tester, and hasher.

```
public SeparateChaining(){
    this(DEFAULT_CAPACITY, DEFAULT_LOAD, DEFAULT_EQUIVALENCE_TESTER);
}
```

The constructor that takes the single argument, *equivalenceTester*, the comparator defining equivalence, creates a new set with a default initial capacity, load, and hasher.

```
public SeparateChaining(Comparator<? super E> equivalenceTester) {
    this(DEFAULT_CAPACITY, DEFAULT_LOAD, equivalenceTester);
}
```

The constructor with the single argument *capacity*, the desired capacity for the set, creates a new set with the given capacity, and the default load, equivalence tester, and hasher.

```
public SeparateChaining(int capacity) {
    this(capacity, DEFAULT_LOAD, DEFAULT_EQUIVALENCE_TESTER);
}
```

The constructor that takes the two arguments *capacity*, the desired capacity, and *equivalenceTester*, the comparator defining equivalence, creates a new set with the provided capacity and equivalence tester, using the default load and hasher.

```
public SeparateChaining(int capacity, Comparator<? super E> equivalenceTester) {
    this(capacity, DEFAULT_LOAD, equivalenceTester);
}
```

The constructor that takes the two arguments *capacity*, the desired capacity, and *load*, the desired load factor, creates a new set with the provided capacity and load, using the default equivalence tester and hasher. It throws an *IllegalArgumentException* when *capacity* < 0.

```

public SeparateChaining(int capacity, double load){
    this(capacity, load, DEFAULT.EQUIVALENCE.TESTER);
}

```

23.5.2 Trivial Accessors

The *getSize* and *isEmpty* methods are inherited from *AbstractCollection*. The *getCapacity* class returns the capacity of the underlying representation for the current hash table to meet the desired load factor.

```

public int getCapacity() {
    return (int) (table.length*targetLoad);
}

```

Correctness Highlights: Since the load factor $\alpha = n/m$, solving for n yields that the capacity is $m \cdot \alpha$ which is the value returned (rounded down to the nearest integer). ■

As in open addressing, the hash function is computed using the **multiplication method**, which has the property that a random value is equally likely to hash to each of the m possible values. See Section 22.4.4 for a further discussion of this hash function including the choice for the constant A .

```

public static final double A = (Math.sqrt(5.0)-1)/2;

```

The *hash* method takes x , an arbitrary integer, and returns an integer between 0 and $m - 1$.

```

int hash(int x) {
    double frac = (x * A) - (int) (x * A);           //fractional part of x*A
    int hashValue = (int) (table.length * frac);    //multiply by m
    if (hashValue < 0)                               //if this is negative add m to get
        hashValue += table.length;                 //hashValue mod m
    return hashValue;
}

```

Since collisions are resolved by placing all elements that hash to the same slot in a list, a secondary hash function is not needed.

23.5.3 Algorithmic Accessors

The internal method *locate* takes *element*, the target, and returns a reference to the chain item for an equivalent element in the set, or *null* if no equivalent element exists.

```

ChainItem<E> locate(E element){
    ChainItem<E> x = table[hash(hasher.getHashCode(element))]; //chain to check
    while (x != null && !(equivalent(x.element, element)))
        x = x.next;
    return x;
}

```

Correctness Highlights: Follows from PLACEMENT and the fact that for each chain item x in $table[hash(hasher.getHashCode(element))]$, $x.element$ is compared to $target$ for equivalence. ■

The method *contains* takes *element*, the target, and returns *true* if and only if an equivalent value exists in this set.

```
public boolean contains(E element) {
    return locate(element) != null;
}
```

The *getEquivalentElement* method takes *target*, the target element, and returns an equivalent element that is in the collection. It throws a *NoSuchElementException* when there is no equivalent element in the collection.

```
public E getEquivalentElement(E target) {
    ChainItem<E> ci = locate(target);
    if (ci != null)
        return ci.element;
    else
        throw new NoSuchElementException();
}
```

23.5.4 Representation Mutators

The internal *resizeTable* method takes *desiredCapacity*, the number of elements that can be placed in this set while maintaining the load factor of *targetLoad*. The hash function depends on the hash table size. Therefore, resizing the hash table implies that elements must be moved according to the size of the new table. This invalidates all active markers for iteration.

While this method could be implemented by calling *newTable.addAll(this)*, we present a more efficient implementation that exploits the fact that all elements in this set are distinct. Thus, the new elements can just be inserted to the front of the appropriate chain in the new hash table, without checking for duplicates.

```
void resizeTable(int desiredCapacity){
    if (hasher.getTableSize(desiredCapacity, targetLoad) != table.length) {
        SeparateChaining<E> newTable =
            new SeparateChaining<E>(desiredCapacity, targetLoad, comp, hasher);
        Locator<E> loc = this.iterator();                                //iterate through all elements
        while (loc.advance()) {
            E e = loc.get();                                              //get next element
            int slot = newTable.hash(hasher.getHashCode(e));              //place at front of its chain
            newTable.table[slot] = new ChainItem<E>(e, newTable.table[slot]);
        }
        this.table = newTable.table;
        version.increment();                                             //invalidate all active locators
        updateResizingBounds();                                         //preserve Threshold
    }
}
```

Correctness Highlights: Since $\alpha = n/m$, for the target load and current *size*, the capacity for table that will satisfy the target load is n/α where n is the *desiredCapacity* for the set, and α is the target load. This value is rounded up to the next integer and used for the new table capacity.

Since, it is known that each element is distinct, placing each element at the front of the chain *slot* maintains PLACEMENT. By the correctness of the *iterator*, PLACEMENT is also preserved. Setting the *size* of the new set, to be equal to that of the old set preserves SIZE. The version number is incremented by *version.increment()* since all active markers must be invalidated. Also, *updateResizingBounds* preserved THRESHOLD by recomputing the low water mark and high water mark. ■

The method *updateResizingBounds* computes the value for *lowWaterMark* and *highWaterMark* that determine when the table should be resized.

```
void updateResizingBounds() {
    lowWaterMark = (int) Math.ceil(table.length*targetLoad*MIN_RATIO);
    highWaterMark = (int) (table.length*targetLoad*MAX_RATIO);
}
```

Correctness Highlights: This method preserves THRESHOLD as long as this method is executed whenever the table is resized or the target load is changed. ■

The internal method *shrinkTableAsNeeded* checks if the hash table is at or below its minimum allowed load, and, if so, resizes within the limitations of the last call to *ensureCapacity*.

```
protected void shrinkTableAsNeeded(){
    if (size ≤ lowWaterMark)
        resizeTable(Math.max(size, minCapacity));
}
```

Correctness Highlights: By definition of the low water mark, PROPORTIONALITY would be violated if the table is not resized when *size* = *lowWaterMark*. If not limited by *minCapacity*, the hash table should be resized so that the target load is met. However, to preserve MINCAPACITY, the capacity is not allowed to go lower than that needed to hold *minCapacity* elements at the target load factor. Thus PROPORTIONALITY and MINCAPACITY are preserved. ■

The internal method *growTableAsNeeded()* checks if the hash table is at or above its maximum allowed load, and, if so, resizes the table to accommodate the current size.

```
protected void growTableAsNeeded(){
    if (size ≥ highWaterMark) //table at maximum capacity
        resizeTable(size);
}
```

Correctness Highlights: Follows from the correctness of *resizeTable*. ■

The internal convenience method *resizeTableAsNeeded* checks to see if the table needs to grow or shrink. The method, which is called whenever the user changes the desired load factor or the desired minimum capacity, preserves PROPORTIONALITY.

```
protected void resizeTableAsNeeded() {
    growTableAsNeeded();
    shrinkTableAsNeeded();
}
```

The *setLoad* method takes *load*, the desired load, and resizes the table so that it has the desired load factor.

```
public void setTargetLoad(double load){
    this.targetLoad = load;
    updateResizingBounds();
    resizeTableAsNeeded();
}
```

Correctness Highlights: THRESHOLD is preserved by *updateResizingBounds*. All other properties are preserved by *resizeTable*. ■

The *ensureCapacity* method takes *desiredCapacity*, the desired capacity, and resizes the table (if needed), so this set can hold *desiredCapacity* elements at the target load factor.

```
public void ensureCapacity(int desiredCapacity){
    minCapacity = desiredCapacity;
    growTableAsNeeded();
}
```

Correctness Highlights: Follows directly from the correctness of *growTableAsNeeded*. ■

The *trimToSize* method adjust the size of the hash table so it is the minimum size needed to maintain the desired load factor.

```
public void trimToSize(){
    minCapacity = size;
    resizeTable(size);
}
```

Correctness Highlights: Follows directly from the correctness of *resizeTable*. ■

23.5.5 Content Mutators

Methods to Perform Insertion

The *add* method takes *element*, the element to be inserted into this set.

```
public void add(E element){
    ChainItem<E> ci;
    if ((ci = locate(element)) != null) {
        ci.element = element;
    }
```

//check for equivalent element
//found equivalent element
//replace by new element

```

else {
    int slot = hash(hasher.getHashCode(element));           //place new element at front of
    table[slot] = new ChainItem<E>(element, table[slot]);    //appropriate chain
    size++;
    growTableAsNeeded();
}
}

```

Correctness Highlights: By the correctness of *locate*, if an equivalent element is in the set, it is found and replaced by the provided element. We now consider when there is no equivalent element in the set. First the element is added to the chain *table[hash(hasher.getHashCode(element))]*. Clearly this maintains PLACEMENT. Since the size of the set has increased by one, incrementing the instance variable *size* preserves SIZE. THRESHOLD and PROPORTIONALITY are preserved by *growTableAsNeeded*, which is called when the size of the set increases. Also *growTableAsNeeded* invalidates all active markers. Finally, MINCAPACITY is maintained since the table size remains the same or increases. ■

Methods to Perform Deletion

The *clear* method removes all elements from the collection.

```

public void clear() {
    Arrays.fill(table, 0, table.length, null);           //preserve Placement
    size = 0;                                             //preserve Size
    shrinkTableAsNeeded();                               //preserve Proportionality and MinCapacity
    version.increment();                                  //invalidate markers
}

```

Correctness Highlights: PLACEMENT is maintained by the call to *Arrays.fill*, since it removes all elements. SIZE is maintained by setting *size* to 0. Finally, *shrinkTableAsNeeded* preserves PROPORTIONALITY and MINCAPACITY. Finally, all active markers are invalidated. ■

For space efficiency, a singly linked list is used for the chains. A consequence of this is that *locate* cannot be used for deletion since the predecessor of the element to be removed is needed. The search is similar to that of *locate*, except that the comparison is always made to the next element in the list to keep track of the predecessor. Also, since there is no head sentinel, a special case is needed for when the element to be removed is held at the front of its chain. The *remove* method takes *element*, the target, and removes an equivalent element (if one exists) from the set. It returns *true* if and only if the element was in the set and therefore removed.

```

public boolean remove(E element) {
    version.increment();                                   //invalidate markers
    int slot = hash(hasher.getHashCode(element));         //compute slot for chain
    ChainItem<E> toRemove = null;                         //set to item if found
    ChainItem<E> ptr = table[slot];                       //ptr traverses the chain
    if (ptr == null)                                       //chain empty which means element
        return false;                                     //is not in the set
    if (equivalent(ptr.element, element)) {               //if element first in its chain
        toRemove = table[slot];
    }
}

```

```

        table[slot] = table[slot].next;                                //have head pointer bypass it
    }
    while (toRemove == null && ptr.next != null) {                    //traverse chain looking one ahead
        if (equivalent(ptr.next.element, element)) {                //if its next
            toRemove = ptr.next;
            ptr.next = ptr.next.next;                                //preserve Placement
        } else
            ptr = ptr.next;                                           //otherwise, move forward in the chain
    }
    if (toRemove != null) {                                          //if found perform needed updates
        size--;                                                       //preserve Size
        toRemove.markDeleted();                                       //preserve InUse
        shrinkTableAsNeeded();                                        //preserve Proportionality and MinCapacity
        return true;
    }
    return false;                                                    //if not found, return false
}

```

Correctness Highlights: By PLACEMENT, if *element* is in the set, then it is in the chain *table[slot]*. The local variable *ptr* is initialized to the first item in the chain, which is *null*, if *table[slot]* is empty. By PLACEMENT, if the initial value of *ptr* is *null*, *element* is not contained in the set and *false* is correctly returned.

If the element is found in the chain, then the chain item holding it is removed from the chain, which preserves PLACEMENT, and *toRemove* references the chain item that is not longer in use. When the element was found, *size* is decremented to preserve SIZE, the chain item *markDeleted* called to *toRemove* preserves INUSE. *shrinkTableAsNeeded* preserves PROPORTIONALITY and MINCAPACITY, and invalidates all active markers. Finally *true* is correctly returned. By PLACEMENT, if there is no equivalent element in chain *table[slot]*, *false* is correctly returned. ■

23.5.6 Locator Initializers

The *iterator* method creates a new marker that is initialized to FORE.

```

public Locator<E> iterator(){
    return new Marker(FORE);
}

```

Correctness Highlights: Follows from the Marker constructor and the fact that an argument of -1 to the Marker constructor corresponds to a marker position just before the “first” element in the set. ■

The *getLocator* method takes *element*, the target, and returns a marker initialized to the first equivalent element in the iteration order. It throws a *NoSuchElementException* when there is no equivalent element in this set.

```

public Locator<E> getLocator(E element) {
    ChainItem<E> node = locate(element);
    if (node == null)

```

```

    throw new NoSuchElementException();
    return new Marker(hash(hasher.getHashCode(element)), node);
}

```

Correctness Highlights: Follows from *locate* and the *Marker* class constructor. ■

23.6 Marker Inner Class

`AbstractCollection<E>.AbstractLocator<E>` implements *Locator<E>*

↑ **Marker** implements *Locator<E>*

We now describe the *Marker* inner class that implements the *Locator* interface. As with direct addressing, since an element can be located in expected constant time, there is no reason to support a tracker unless the application requires worst-case constant time access. The *Marker* class has an instance variable, *slot* that stores the slot of the table where the marker is located, and *chainLoc* references the chain item where the marker is located.

```

private int slot;                                //slot for current marker
private ChainItem<E> chainLoc;                  //location within chain table[slot]

```

For separate chaining, the location that is marked is a combination of the slot number of the hash table and a reference into the current chain (when not between chains). As with direct addressing, the iteration order for separate chaining has no relation to the order in which the elements were added to the set. The ordering used will be to iterate through the non-empty chains from slot 0 to slot $m - 1$. Again we think of FORE as being slot “-1” and AFT as being slot “ m .” More formally, let $s_0, \dots, s_c$ be the indices of the non-empty chains. The iteration order that is defined by the marker is $\langle \text{table}[s_0], \text{table}[s_1], \dots, \text{table}[s_c] \rangle$ where $\text{table}[s_i]$ contains the elements held in chain $\text{table}[s_i]$ in the order they occur in the chain (which is opposite of their insertion order into the chain).

As in direct addressing (Chapter 21), when $\text{table}[i] = \text{null}$, we define $\text{pred}(i)$ to be the largest integer j such that $j < i$ and $\text{table}[j]$ is not *null*. When an element is removed through a marker, and then *advance* is called, the marker should be at the element in the iteration order after the removed element. While the implementation could directly perform this update, that could be inefficient. Hence, when removing the last element from a slot, our implementation physically leaves the marker at the current slot. However, logically, the marker holding i is at $\text{pred}(i)$. We use the following abstraction function to capture this behavior

Abstraction Function: Let *loc* be a marker instance. The abstraction function

$$AF(loc) = \begin{cases} loc.chainLoc.element & \text{when } loc.chainLoc \neq null \\ \text{last element in chain } table[pred(loc.slot)] & \text{otherwise.} \end{cases}$$

In our correctness arguments, we argue that when no critical mutation has occurred, this abstraction function is consistent with the iteration order given above and the semantics required of the marker as described in Section 5.8. Whenever *chainLoc* references a chain item x that is not in use, the critical mutation of removing the element held in x must have been called either directly or through a different marker. At this point, iteration cannot be guaranteed to continue correctly since there is no mechanism to determine the current location for the marker. (Section 23.1 discusses a

way such a mechanism could be supported, if desired.) When this situation occurs, our implementation logically sets the marker location to the start of the current chain, which might cause some elements to be repeated in the iteration sequence. The other option would be to move to the start of the next chain, which could cause some elements to be skipped.

The constructor that takes *s*, the slot holding the chain item to mark, and *chainLoc*, a reference to the chain item itself. It requires *s* is FORE, AFT, or a valid slot in the hash table.

```
Marker(int s, ChainItem<E> chainLoc){
    slot = s;
    this.chainLoc = chainLoc;
    versionNumber = version.getCount();
}
```

A second constructor takes *s*, the slot to mark, and initializes *slot* to the given value, and initializes *chainLoc* to *null*.

```
Marker(int s){
    this(s, null);
}
```

The marker provides an accessor, *inCollection*, that returns *true* if and only if the marker is at an element of the collection.

```
public boolean inCollection(){
    checkValidity(); //check if marker has been invalidated
    if (chainLoc != null && chainLoc.isDeleted())
        chainLoc = null; //move to start of current chain
    else if (slot != FORE && slot != table.length && chainLoc == null)
        retreat();
    return (slot != FORE && slot != table.length);
}
```

Correctness Highlights: By INUSE and the correctness of the ChainLoc *isDeleted* method, when *chainLoc* references a chain item that is no longer in use, the marker is reset to the front of the current chain. Otherwise, if *slot* is between 0 and *table.length - 1*, and *chainLoc* is *null*, then the marker must be updated to *pred(slot)*. The rest follows from the correctness of *retreat* which performs the required marker update. ■

The marker also provides an accessor, *get*, that returns the element stored at the marker location. It throws a *NoSuchElementException* when the marker is logically before the first element or after the last element in the set.

```
public E get() {
    if (!inCollection())
        throw new NoSuchElementException();
    return chainLoc.element;
}
```

Correctness Highlights: Follows from the correctness of *inCollection* and the abstraction function. ■

The *advance* method moves the marker to the next location (if there is one). This method throws an *AtBoundaryException* when the marker had already advanced past the end of the collection via

an earlier call to either *advance* or *next*. It throws an *AtBoundaryException* when the locator is at AFT since there is no place to advance.

```

public boolean advance(){
    checkValidity();                //check if marker has been invalidated
    if (slot == table.length)       //marker is at AFT
        throw new AtBoundaryException("Already after end.");
    if (chainLoc == null || chainLoc.next == null) {           //move to next slot
        while (++slot < table.length && table[slot] == null); //non-null slot
        if (slot == table.length)                             //if at AFT
            return false;                                     //return false
        chainLoc = table[slot];                               //otherwise, set chainLoc
    } else {                                                  //move to next element in chain
        chainLoc = chainLoc.next;
    }
    return true;
}

```

Correctness Highlights: If the marker is at AFT (i.e., at slot m), the exception is properly thrown. We now consider the following two cases.

***chainLoc* is null or references the last element in a chain:** In this case, the marker is logically just after the last element in the current chain. The marker should thus be set to the first element in the next non-empty chain. If slots $slot + 1, \dots, table.length - 1$ are empty, the marker is properly moved to AFT, and *false* is returned. Otherwise *true* is returned.

***chainLoc* references an element in the chain that is not the last element:** In this case, the marker is correctly moved to the next item in the chain, and *true* is properly returned. ■

Since *retreat* requires moving backwards in the iteration order, we provide two methods to support it. The *toEnd* method takes s , the desired slot. It moves the marker to the last chain item in slot s . The time complexity for this method is proportional to the length of the chain at slot s . Thus, if *retreat* is used frequently, the load factor should be kept small to ensure that the expected chain length is small.

```

private void toEnd(int s){
    ChainItem<E> ptr = table[s];
    while (ptr.next != null)
        ptr = ptr.next;
    chainLoc = ptr;
}

```

Correctness Highlights: Follows from the proper construction of the chain. ■

The *toPredecessor* method takes s , the desired slot. It moves the marker to the chain item in slot s that precedes its current location, or *null* if the current location is at the first chain item in slot s . This method requires that *chainLoc* references a chain item in $table[s]$. The time complexity for this method is also proportional to the length of the chain at slot s .

```

private boolean toPredecessor(int s) {
    if (slot != table.length) {
        //marker not at AFT
    }
}

```

```

ChainItem<E> ptr = table[s];           //first item in current list
if (chainLoc != ptr) {                 //chainLoc not first in slot s
    while (ptr.next != chainLoc)
        ptr = ptr.next;
    chainLoc = ptr;
    return true;
}
}
chainLoc = null;                       //chainLoc is first in slot s
return false;
}

```

Correctness Highlights: By the requirement on this method, and the construction of the chain, this method satisfies its specification. ■

The *retreat* method moves the marker to the previous location. It returns *true* if and only if the updated value for *slot* is one that is in use. The user program can use this return value to recognize when a marker has moved before the start of the set. It throws an *AtBoundaryException* when the marker is at *FORE* because retreating from *FORE* is impossible.

```

public boolean retreat(){
    checkValidity();                     //check if marker has been invalidated
    if (slot == FORE)                    //marker at FORE
        throw new AtBoundaryException("Already before front.");
    if (toPredecessor(slot))
        return true;                     //there was a previous element in the chain
    while (--slot ≥ 0 && table[slot] == null); //move to previous non-empty chain
    if (slot == FORE)
        return false;
    toEnd(slot);                         //position chainItem at end of chain
    return true;
}

```

Correctness Highlights: If the marker is at *FORE*, the exception is properly thrown. Otherwise, by the correctness of *toPredecessor*, *chainLoc* is moved to the previous element in the chain, or *null* if the current location was the first element in its chain. We now consider the following two cases.

***chainLoc* is not null:** In this case, *toPredecessor* already made the needed update, and *true* is properly returned.

***chainLoc* is null:** This case only occurs when the current marker location was the first element in its chain. In this case, the marker should be set to the last element in the first non-empty chain in the sequence $slot - 1, \dots, 0$. If all of these chains are empty, the marker is properly moved to *FORE*, and *false* is returned. Otherwise, the marker is properly moved to the end of the first non-empty chain found, and *true* is returned.

As specified in the abstraction function, this method will place the marker at the previous element in the iteration order. If the marker is currently at the first element in the iteration order, then it will set *slot* to -1 which corresponds to the marker being at *AFT*. ■

To implement the Java *Iterator* interface, we provide the *hasNext* method which returns *true* if there is some element after the current marker location. Since *hasNext* is typically followed by *next* or *advance*, our implementation performs the optimization of moving the marker up to the end of the chain immediately preceding the one that holds the next element in the iteration order.

```
public boolean hasNext(){
    checkValidity();                                //check if marker has been invalidated
    if (chainLoc != null && chainLoc.next != null)    //next element in current chain
        return true;
    chainLoc = null;                                //at end of current chain item
    while (++slot < table.length && table[slot] == null); //find next non-empty chain
    slot--;                                         //move back to "end" of the preceding empty chain
    return (slot < table.length-1);
}
```

Correctness Highlights: We now consider the following three cases.

***chainLoc* is null or references the last element in a chain:** In this case, the marker is logically just after the last element in the current chain. The next element in the iteration order is that in the next non-empty chain. The while loop moves *slot* to the first non-empty chain among $slot + 1, \dots, table.length - 1$. However, *hasNext* should not advance the marker, so *slot* is decremented so that the logical position of the marker does not change, but the work of finding the next non-empty chain need not be repeated. In this case, the return value from *hasNext* should be *false* only when no non-empty chain was found. When this situation occurs, *slot* will have value *table.length* after the while loop. So *table.length* - 1, after *slot* is decremented. Thus, the correct value is returned.

***chainLoc* references an element in the chain that is not the last element:** In this case, *true* is properly returned.

Marker is currently at AFT: In this case, *chainLoc* is null, and the while loop does not execute since $slot = table.length$. Finally observe that the correct value of *false*, is returned. ■

The *remove* method removes the item at the marker. It throws a *NoSuchElementException* when the marker is logically at FORE or AFT. After this method the marker is logically at the previous item in the iteration order.

```
public void remove(){
    if (!inCollection())                                //calls checkValidity and adjusts marker
        throw new NoSuchElementException();
    ChainItem<E> toRemove = chainLoc;                    //reference to element to remove
    toPredecessor(slot);
    if (chainLoc == null) {                               //no predecessor
        table[slot] = table[slot].next;                  //remove it
        chainLoc = null;                                //move marker to the end of
        slot--;                                         //prior slot
    } else {                                             //chainLoc references the predecessor
        chainLoc.next = chainLoc.next.next;              //remove it
    }
    shrinkTableAsNeeded();                               //preserve Proportionality
    toRemove.markDeleted();                             //preserve InUse
}
```

method	expected complexity	worst-case complexity
<i>advance</i>	$O(1)$	$O(n)$
<i>get()</i>	$O(1)$	$O(n)$
<i>hasNext()</i>	$O(1)$	$O(n)$
<i>inCollection()</i>	$O(1)$	$O(n)$
<i>next()</i>	$O(1)$	$O(n)$
<i>remove()</i>	$O(1)$	$O(n)$
<i>retreat()</i>	$O(1)$	$O(n)$

Table 23.2 Summary of the expected time complexities for the public locator methods when using the open addressing data structure when $m \leq c \cdot n$ for some constant c .

```
size--; //preserve Size
updateVersion(); //ensures that this locator is still valid
}
```

Correctness Highlights: By the correctness of *inCollection*, either a *NoSuchElementException* is thrown, or *chainLoc* is moved to the previous element in the chain, or *null* if the current location was the first element in its chain. We now show that *PLACEMENT* is preserved. Consider the following two cases.

***chainLoc* is *null*:** This case only occurs when the current marker location was the first element in its chain. In this case, this chain item is spliced out of the list by having *table[slot]* bypass it. Finally, the proper location for the marker is just before the current slot. According to the abstract function, this is achieved by setting *chainLoc* to *null*, and decrementing *slot*, which places the marker at the end of the previous chain.

***chainLoc* is not *null*:** In this case, *toPredecessor* has already updated the marker to the predecessor, so the only change needed is to bypass the chain item that is to be removed.

PROPORTIONALITY is preserved by *shrinkTableAsNeeded*, *INUSE* is preserved by the Chain-Item *markDeleted* method, and *SIZE* is preserved by decrementing *size* by 1. *PLACEMENT* is not affected since no chain items are moved. Finally, to allow iteration to continue through this marker, the version number is updated. ■

Set

23.7 Performance Analysis

The asymptotic time complexities of all public Set methods for separate chaining are identical to those of open addressing as shown in Table 22.2, and the asymptotic time complexities for all of the public methods of the Marker class are given in Table 23.2. Throughout our analysis, we make the assumption that the *hashCode* takes constant time, and that $m \leq c \cdot n$ for some constant c . Thus $\alpha = n/m$ is a constant.

For any method that must find an existing element, the time complexity is dominated by time complexity of the *locate* method, or in the case of *remove*, the similar search for chain item (or slot) that references the element to remove. Observe that with n elements divided among m slots,

The methods *add*, *contains*, *getEquivalentElement*, *getLocator*, and *remove* take expected constant time since they require constant time in addition to the time to locate the desired item. The *accept*, *clear*, *toArray*, and *toString* methods iterate through each chain of table which takes $O(m+n)$ time since there are m references in the hash table to follow, and n references within the chains. The internal *resizeTableAsNeeded* method takes $O(m+n)$ time since constant time is spent to *add* each element to the new table during the process of iterating through all of the chains. Since we assume $m = O(n)$, *clear*, *toArray*, *toString*, and *resizeTableAsNeeded* take worst-case $O(n)$ time. Since *ensureCapacity(x)* must initialize all x slots to *null*, it takes $O(x)$ time. Finally, *addAll(c)* and *retainAll(c)* take constant time for each element in c .

$$2n + m = 2n + \frac{n}{\alpha} = n \left(2 + \frac{1}{\alpha} \right),$$

We now analyze the methods in the Marker inner class. The *advance*, *hasNext*, and *next* methods take expected constant-time since they must find the next slot in use. Under the assumption each element is equally likely to hash to any slot, the expected number of slots required to reach the next element is m/n , the expected time complexity is $m/n = O(1)$. The *toEnd* and *toPredecessor* methods have time complexity $O(\alpha)$, which is constant under the assumptions we have made. Thus the *get*, *remove*, and *retreat* methods take expected constant time since the dominant cost is that of *toEnd* and *toPredecessor*. Observe that if *retreat* is used to iterate through the collection, then the overall expected cost is $O(\alpha^2 \cdot n)$.

23.8 Quick Method Reference

```

p. 327 SeparateChaining()
p. 327 SeparateChaining(Comparator<? super E> equivalenceTester)
p. 327 SeparateChaining(int capacity)
p. 327 SeparateChaining(int capacity, Comparator<? super E> equivalenceTester)
p. 327 SeparateChaining(int capacity, double load)
p. 327 SeparateChaining(int capacity, double load, Comparator<? super E> equivalenceTester)
p. 326 SeparateChaining(int capacity, double load, Comparator<? super E> equivalenceTester,
    Hasher<? super E> hasher)
p. 98 void accept(Visitor<? super E> v)

```

- p. 331 void **add**(E element)
- p. 100 void **addAll**(Collection<? extends E> c)
- p. 100 void **clear**()
- p. 97 boolean **contains**(E value)
- p. 99 void **ensureCapacity**(int capacity)
- p. 96 int **getCapacity**()
- p. 96 Comparator<? super E> **getComparator**()
- p. 98 E **getEquivalentElement** (E target)
- p. 333 Locator<E> **getLocator**(E element)
- p. 96 int **getSize**()
- p. 96 boolean **isEmpty**()
- p. 333 Locator<E> **iterator**()
- p. 332 boolean **remove**(E element)
- p. 100 void **retainAll**(Collection<E> c)
- p. 331 void **setTargetLoad**(double load)
- p. 97 Object[] **toArray**()
- p. 97 E[] **toArray**(E[] array)
- p. 98 String **toString**()
- p. 99 void **trimToSize**()

SeparateChaining Internal Methods

- p. 97 int **compare**(E e1, E e2)
- p. 97 boolean **equivalent**(E e1, E e2)
- p. 330 void **growTableAsNeeded**()
- p. 328 int **hash**(int x)
- p. 328 ChainItem<E> **locate**(E element)
- p. 329 void **resizeTable**(int desiredCapacity)
- p. 330 void **resizeTableAsNeeded**()
- p. 330 void **shrinkTableAsNeeded**()
- p. 99 void **traverseForVisitor**(Visitor<? super E> v)
- p. 330 void **updateResizingBounds**()
- p. 98 void **writeElements**(StringBuilder s)

SeparateChaining.Marker Public Methods

- p. 336 boolean **advance**()
- p. 335 E **get**()
- p. 338 boolean **hasNext**()
- p. 101 void **ignoreConcurrentModifications**(boolean ignore)
- p. 101 void **ignorePriorConcurrentModifications**()
- p. 335 boolean **inCollection**()
- p. 101 E **next**()
- p. 338 void **remove**()
- p. 337 boolean **retreat**()

SeparateChaining.Marker Internal Methods

- p. 335 *Marker*(int s)
- p. 335 *Marker*(int s, ChainItem<E> chainLoc)
- p. 101 void **checkValidity**()
- p. 336 void **toEnd**(int s)
- p. 336 boolean **toPredecessor**(int s)
- p. 101 void **updateVersion**()

Chapter 24

Priority Queue ADT

package collection.priority

Collection<*E*>

↑ *PriorityQueue*<*E*>

A priority queue is an untagged algorithmically positioned collection of comparable elements in which there can be equivalent elements. The primary methods are to add an element, find a maximum priority element, and remove a maximum priority element. In addition, one can remove an element or change the priority of an element through a locator. However, there is no efficient general search method.

The iteration order is arbitrary, with no dependence on the ordering among the elements. In fact, the iteration order may change across different iterations. The only requirement is that each persistent element is seen exactly once in a completed iteration. One can view the *PriorityQueue* interface as a limited version of the *OrderedCollection* interface (Chapter 29). Because a priority queue need not support a general purpose search, or provide iteration based on the ordering of the elements, the data structures to implement it are generally more efficient (in space and time) than those for an ordered collection.

24.1 Case Study: Huffman Compression

Data Compression is the process of representing a document* using as few bits as possible. We consider **lossless compression**, in which the original document can be reconstructed from the compressed version. Compression is used for many tasks, including reducing the number of bits required to transmit a photo, and creating an archive of a directory in a file system. A **codeword** is a sequence of bits that is used to represent each “character” in the document, and a **variable length code** is one in which the codewords vary in length. Intuitively, more frequently used characters should have short codewords, whereas less frequently used characters should have longer codewords. A **prefix-free code** is a set *S* of codewords in which no codeword in *S* is a prefix of another codeword in *S*. In 1952, David Huffman [88] developed a method of choosing a codeword for each character of a document that can be proven to encode the document in the fewest number of bits possible for any prefix-free code. The resulting set of codewords is called a **Huffman code**.

The algorithm to construct Huffman codes builds a binary tree from the bottom up to represent the optimal prefix code. The advantage of using a prefix-free code for compression is that it is not necessary to provide a delimiter that indicates where one code word ends and the next begins. Eliminating delimiters saves additional space in the encoded file. Figure 24.1 shows a sample Huffman code and the corresponding tree. As an example, the word “faced” is represented with the code “01110001010111.” To decode it, start at the root, and process characters until a leaf is

*We are using document in a very broad way to include any multimedia document.

	a	b	c	d	e	f	g
frequency (in hundreds)	20	14	12	16	25	8	5
Huffman code	00	110	010	111	10	0111	0110

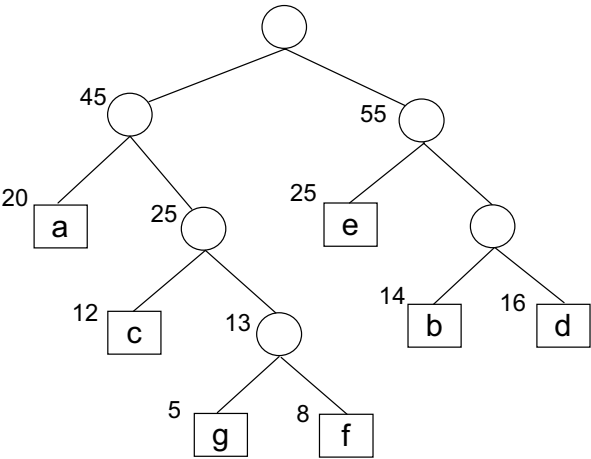


Figure 24.1
The frequency for each character in a file, along with the Huffman code, and the Huffman code binary tree used to construct the code, and to decode. In the tree, the left branch corresponds to a 0 in the codeword, and the right branch corresponds to a 1 in the codeword. Next to each node is the corresponding frequency. In constructing the tree, we use the convention that the lower priority node is the left child.

reached. In this case the leaf “f” is reached after processing “011.” Then return to the root, and repeat the process with the remaining bits.

A key step in the process of building the tree to represent the code is to find the two least frequent characters and make them children of the same internal node of the tree, which virtually combines them into a new character. Each character has an associated frequency. When character a with frequency f_a is combined with character b with frequency f_b , the new virtual character has frequency $f_a + f_b$. Such virtual characters can be combined with other actual or virtual characters to create additional virtual characters. In the tree of Figure 24.1, the virtual characters correspond to the internal nodes. The frequency for each character and virtual character is shown next to the corresponding node.

Building the Huffman code binary tree requires a second data structure that stores each actual and virtual character in such a way that the character with the lowest frequencies can be efficiently found and removed for combining into a new virtual character, which must then be efficiently inserted into that data structure. It is also necessary. Observe that if one defines the priority of a character a to be higher than that of character b exactly when $f_a < f_b$, then the primary methods this data structure must support are to remove a highest priority element from the collection and to add a new element to the collection. The priority queue is a perfect match for these needs.

24.2 Interface

The following methods are those added to those inherited from the *Collection* interface. The methods that return a locator are overridden to instead return a priority queue locator that supports an additional method to update the priority of an element (see Section 24.3).

PriorityQueue(): Creates a priority queue that uses the default comparator.

PriorityQueue(Comparator comp): Creates a priority queue that uses the provided comparator.

E extractMax(): Removes and returns a highest priority element. It throws a *NoSuchElementException* when this collection is empty.

PriorityQueueLocator<E> getLocator(E element): Returns a priority queue locator that has been set to the given element. It throws a *NoSuchElementException* if there is no equivalent element in this collection.

E max(): Returns a highest priority element. It throws a *NoSuchElementException* when this collection is empty.

Critical Mutators for PriorityQueue: *add, addAll, addTracked, clear, extractMax, remove, retainAll*

24.3 Priority Queue Locator Interface

package collection.priority

Locator<E>

↳ **PriorityQueueLocator<E>**

We extend the *Locator* interface by adding a method to update the priority of an element. We let e be the element currently referenced by the locator.

void decreasePriority(E element): Replaces the element associated with this priority queue locator by the given lower priority element. This method requires that the given parameter is less than e , or that e is the parameter being passed and its value has been mutated to have a lower priority than it had previously. That is, it is acceptable practice to mutate the element to have a lower priority and then immediately call *decreasePriority* to restore the properties of the priority queue.

void increasePriority(E element): Replaces the element associated with this priority queue locator by the given higher priority element. This method requires that the given parameter is greater than e , or that e is the parameter being passed and its value has been mutated to have a greater priority than it had previously. That is, it is acceptable practice to mutate the element to have a greater priority and then immediately call *increasePriority* to restore the properties of the priority queue.

void update(E element): Replaces the element associated with this priority queue locator by the given element. This method requires that the parameter is not the same object as e . This is necessary because the method compares the old and the new values to decide whether the priority must be decreased or increased. Therefore, it would be an error to mutate the element referenced by the locator and then call *update*.

```
public void update(E element);
```

Critical Mutators for PriorityQueueLocator: *decreasePriority*, *increasePriority*, *remove*, *update*

24.4 Selecting a Data Structure

To aid in the process of selecting the best *PriorityQueue* data structure, consider which of the following properties are most important for the desired application. The properties are listed from the most to least significant in terms of how much they should affect the decision process.

Acceptability of a self-organizing data structure: All of the primary methods for a binary heap and a leftist heap have either constant or logarithmic worst-case time complexity. In contrast, the pairing heap and Fibonacci heap make major changes to the organization of the data structure as part of their implementation for some methods. While both of the latter data structures have very efficient amortized cost for the primary methods, the actual costs of some of the other methods are significant.

Time complexity for merge: A major distinction between the data structures is the time complexity for the *merge* method. It takes linear time for a binary heap, logarithmic time for a leftist heap and pairing heap, and constant time for a Fibonacci heap.

Space usage: The space usage for the binary heap is excellent. In particular for untracked implementations where the constructor provides the appropriate size, the space requirement is just a single reference per element. The leftist heap and pairing heap (when not threaded)

also have fairly good space usage, but still about 3-4 times more than a binary heap. The Fibonacci heap has the highest space usage.

Amortized constant-time insert: The Fibonacci heap is the only data structure that has constant worst-case and amortized cost. The actual cost for the pairing heap is constant, but the amortized cost is logarithmic.

Amortized constant-time method to increase an element's priority through a locator: The Fibonacci heap is the only data structure that has constant amortized cost. The actual cost for the pairing heap is constant, but the amortized cost is logarithmic.

Simplicity: The binary heap is an extremely simple data structure. The Fibonacci heap has the most complex methods, which also adds to the constant hidden within the asymptotic notation.

24.5 Terminology

In addition to the definitions given in Chapter 7, we use the following definitions throughout our discussion of the PriorityQueue ADT and the data structures that implement it. All of the priority queue data structures are composed of nodes logically organized as a rooted tree, or as a forest of rooted trees.

- When element $e1$ is less than element $e2$ with respect to the comparator associated with this priority queue, we write $e1 < e2$. Likewise, for the $\leq$, $>$, $\geq$, and $=$ operators between elements $e1$ and $e2$.
 - For two nodes x_1 and x_2 , we say that x_1 has a higher (respectively, lower) priority than x_2 when the element referenced by x_1 has a higher (respectively, lower) priority than that referenced by x_2 .
 - When $e1 > e2$, we say that element $e1$ has a *higher priority* than element $e2$.
 - A *highest priority element* is one for which there is no element in the collection with a higher priority.
 - We say that a tree, in which each node has priority at least as high as its children, is **heap-ordered**.
 - We say that heap node w is a **descendant** of heap node x when either $w = x$, or w can be reached by following a sequence of child references from x .
 - For heap node x we define $T(x) = \{w \mid w \text{ is a descendant of } x\}$.
-

24.6 Competing ADTs

We briefly discuss alternative ADTs that may be appropriate in a situation when the PriorityQueue ADT is being considered.

OrderedCollection ADT: If the application must efficiently search for elements, or requires the iteration order to be based on the ordering of the elements, the OrderedCollection ADT is a better choice.

TaggedPriorityQueue ADT: If it is more natural for the application to define a priority based on a tag associated with the element (versus by defining a comparator for the elements held in the collection) and it is uncommon for multiple elements to have the same tag, then the TaggedPriorityQueue ADT is more appropriate.

TaggedBucketPriorityQueue ADT: If it is more natural for the application to define a priority based on a tag associated with the element (versus by defining a comparator for the elements held in the collection) and it is common for multiple elements to share the same tag, then the TaggedBucketPriorityQueue ADT is more appropriate.

24.7 Summary of Priority Queue Data Structures

Table 24.2 provides a high-level summary of the trade-offs between the priority queue data structures we present. We also include the skip list (Chapter 38) data structure in the table as an example of an ordered collection data structure that efficiently supports *max*, *extractMax*, *advance*, and *re-treat*[†]. The constant hidden within the asymptotic notation is generally smaller for the priority queue data structures than the skip list. We now discuss the kinds of applications for which each of them is best suited.

BinaryHeap: The binary heap is a very simple data structure that has worst-case logarithmic cost for *add*, *extractMax*, and *update* (through a locator). However, the lead constant within the asymptotic notation is small, so all of the methods run quite quickly. Also the space usage is very efficient, especially if the required size is passed to the constructor. The main drawback is that merging two binary heaps takes linear time. Also, increasing the priority of a tracked element takes logarithmic time.

LeftistHeap: The leftist heap is a fairly simple implementation that supports *merge* in logarithmic time. A leftist heap can be threaded to support faster iteration, or non-threaded. We provide only a non-threaded leftist heap implementation, but a threaded version could be implemented as illustrated for a pairing heap. The primary cost of the threading would be the increased space due to two additional references per element. There would also be a small amount of overhead required to maintain the extra references. In the threaded implementation, all of the methods would have the same time complexity as for the binary heap, except for *merge* which is improved from linear to logarithmic time, and *addAll* which changes from constant time per new element to logarithmic time per new element. In the non-threaded implementation, iteration is limited and less efficient than the threaded implementation.

PairingHeap: The pairing heap is a simple **self-organizing** data structure in which the amortized cost for *add*, *merge*, and *remove* through a tracker are all logarithmic. The insert and merge methods have worst-case constant cost. When an element is removed, the data structure is re-structured, leading to a worst-case linear cost. Increasing the priority for a tracked element has worst-case constant cost, but the amortized cost of this operation is logarithmic due to

[†]Although a skip list does have an *extractMax* method, it can be implemented to run in constant expected time since the maximum is found in constant time, and removing an element once it has been located has expected constant cost.

Key								
⦿	Excellent							
○	Very Good							
⊖	Good							
●	Fair							
•	Method does nothing							
⊗	Unsupported method							
Method		BinaryHeap (not tracked)	BinaryHeap (tracked)	LeftistHeap (not threaded)	LeftistHeap (threaded)	PairingHeap	FibonacciHeap	SkipList (OrderedCollection)
Priority Queue Methods	add(e)	○	○	○	○	○	⦿	⊖
	addAll(c), per element in c	⦿	⦿	○	○	○	⦿	⊖
	addTracked(e)	○	○	○	○	○	⦿	⊖
	clear(), per element	⦿	⦿	⦿	⦿	⦿	⦿	⦿
	contains(e)	●	●	●	●	●	●	⊖
	ensureCapacity(), trimToSize()	●	●	•	•	•	•	•
	extractMax()	○	○	○	○	○	○	⦿
	getEquivalentElement(e), getLocator(e)	●	●	●	●	●	●	⊖
	max()	⦿	⦿	⦿	⦿	⦿	⦿	⦿
	merge(pq1, pq2)	●	●	○	○	○	⦿	●
Other Issues	remove(e)	●	●	●	●	●	●	⊖
	retainAll(c), per element in c	○	○	○	○	○	○	⦿
	toArray(), toString(), per element	⦿	⦿	⦿	⦿	⦿	⦿	⦿
	accept(visitor), per element	⦿	⦿	⦿	⦿	⦿	⦿	⦿
	space usage	⦿	○	⊖	⊖	⊖	●	○
Locator Methods	very small constant hidden in O()	✓	✓	✓	✓			
	amortized					✓	✓	
	randomized							✓
	advance(), get()	⦿	⦿	⊗	⦿	⦿	⦿	⦿
	hasNext(), next()	⦿	⦿	○	⦿	⦿	⦿	⦿
	remove()	○	○	○	○	○	○	⦿
	retreat()	⦿	⦿	⊗	⦿	⦿	⦿	⦿
	update(o) - increase priority	○	○	○	○	○	⦿	⊖
	update(o) - decrease priority	○	○	○	○	○	○	⊖

Time Complexity

⦿ O(1) time

○ O(log n) worst case

⊖ O(logn) expected time

● O(n) time

Space Usage

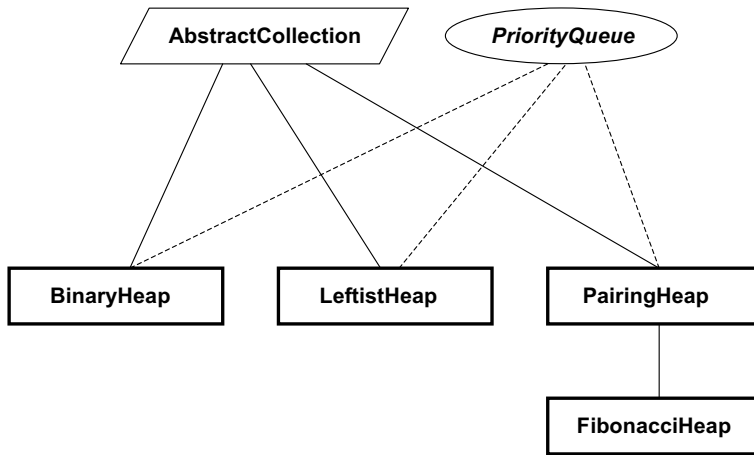
⦿ ≈ n to 2n

○ ≈ 3n to 4n

⊖ ≈ 5n to 7n

● ≈ 10n

Table 24.2 Trade-offs among the data structures that implement a priority queue, as well as the skip list data structure. Technically, when the dynamic array is used within a binary heap, the cost shown for adding or removing an element is amortized over the cost of resizing the array. However, since the reorganization required in resizing is not inherent in the binary heap data structure, we do not view its time complexity as amortized. If the array used within the binary heap is allocated to an appropriate size, the array will not be resized.

**Figure 24.3**

The class hierarchy for the priority queue data structures presented here.

an increase in the potential. Intuitively, each *add* method is charged to compensate for the future reorganization cost. To decrease the priority of a tracked element, the amortized cost is logarithmic, although the worst-case cost is linear. The space usage of the pairing heap is similar to that of the leftist heap. While we provide only a threaded version of a pairing heap, a non-threaded version could be created as illustrated in the leftist heap implementation.

FibonacciHeap: The Fibonacci heap is a more complex self-organizing data structure. It has *amortized* and worst-case costs for *add* and *merge* are constant. Increasing the priority of an element has amortized constant cost, but a worst-case logarithmic cost. Removing an element or lowering the priority of an element has logarithmic amortized cost, but the worst-case cost can be linear. It is a more complex data structure, so unless the priority is often increased, the overall time complexity is higher than that of the pairing heap. However, using a Fibonacci heap yields the asymptotically fastest implementation of Prim's minimum spanning tree algorithm (Section 57.3) and Dijkstra's shortest path algorithm (Section 57.2).

Figure 24.3 shows the class hierarchy for the data structures presented in this book for the Priority Queue ADT.

Design Notes: We use the leftist heap to illustrate two programming techniques that could be applied to other data structures, including the PairingHeap (Chapter 27) and FibonacciHeap (Chapter 28).

- Instead of threading the nodes to support iteration, we use a visiting iterator (Section 8.5). As illustrated in the pairing heap (Chapter 27), we could instead thread the elements. The cost of the threading is two additional instance variables per node and a negligible cost when inserting and removing elements. One disadvantage of the visiting iterator is that only the *next* and *hasNext* methods are supported.
- A truly mergeable leftist heap is supported. While it would be easy to obtain a linear time *merge*, a benefit of a leftist heap over a binary heap is that *merge* can be implemented in logarithmic time. We introduce an instance variable *self* with each leftist heap that initially references itself. Then, if leftist heap *h1* and leftist heap *h2* are merged so that *h1.root* references the resulting heap, by setting *h2.self = h1.self*, then all elements that were in *h2* will be

properly identified as belonging to $h1$. The only other change needed is for the inherited *size* instance variable to be accessed through *self*. See Section 6.8 for further discussion.

Although we choose the leftist heap to illustrate both of these techniques, they could be applied to the other data structures since there is nothing unique about the leftist heap with respect to these techniques.

24.8 Further Reading

Williams [157] first introduced the binary heap and used it as the workhorse of the **heapsort** algorithm (Section 11.4.3) in which all elements are inserted into a binary heap, and then *extractMax* is used to remove the elements in sorted order. Floyd [55] presented a linear time method to convert an arbitrary array into a legal binary heap, allowing *addAll* to be implemented in linear time.

A well-studied extension of the binary heap is a **k -ary heap**, in which each heap node has k children. The standard binary heap is a 2-ary heap. As for a binary heap, a k -ary heap can use an array as its internal representation. The advantage is that the depth of the heap is reduced from $\log_2 n$ to $\log_k n$. The time complexity to insert an element or increase the priority of an element (once located) in a k -ary heap is $O(\log_k n) = O(\log n / \log k)$, which decreases as k increases. However, the time complexity to remove an element or increase the priority of an element (once located) requires comparing the priority of all k children to determine which has the highest priority. Thus the time complexity of these methods is $O(k \log_k n) = O\left(\frac{k}{\log_2 k} \cdot \log n\right)$, which grows with k .

Crane [43] introduced the leftist heap. The textbooks of Knuth [97] and Tarjan [146] are also good resources for discussions of leftist heaps. The pairing heap was introduced by Fredman et al. [64]. Stasko and Vitter [142] performed experiments comparing a variety of priority queue implementations including a two pass variant of a pairing heap which experimentally was more efficient. Moret and Shapiro [116] performed experiments leading to the conclusion that using pairing heaps within Prim's minimum spanning algorithm leads to a more efficient solution than using binary heaps, Fibonacci heaps, or a variety of other priority queue data structures. Additional experimental evaluations between Fibonacci heaps and pairing heaps were performed by Liao [104]. While it was conjectured for a time that merging two pairing heaps and increasing the priority of an element had amortized constant cost, in 1999, Fredman [61] disproved this conjecture, showing that increasing the priority of an element takes more than constant amortized time. More recently, Pettie [124] improved the amortized analysis.

Fredman and Tarjan [65] introduced the Fibonacci heap, and explored its application to graph algorithms, including single-source shortest paths, all-pairs shortest paths, and the minimum spanning-tree problem. (See Chapter 56.)

Some heap variations not presented here are the **binomial heap** introduced by Vuillemin [153] and further studied by Brown [33, 34]. The Fibonacci heap is an extension of the binomial heap that improves the efficiency of any method that does not involve deletion to have constant amortized cost. The pairing heap is one of a variety of self-adjusting heaps. Other self-adjusting heap data structures include the **relaxed heap** [48], and **self-adjusting heap** [141].

The **soft heap** is an interesting variation that randomly increases the priority for a small percentage of the elements so that all methods have expected constant amortized cost. There are other priority queue implementations that are designed for applications where each priority is drawn from a small set of integers [152, 66, 148].

A variation of the PriorityQueue ADT is the **double-ended priority queue**, which has primary methods *add*, *min*, *max*, *extractMin*, and *extractMax*. One could implement a double-ended priority queue as two priority queues where the comparator for one is the complement of the comparator for the other. By having each element in each priority queue hold a tracker to its “mate” in the other, all four of the above methods could be implemented in logarithmic time. However, the space usage of such a solution can be reduced by a more specialized data structure. Data structures explicitly designed for a double-ended priority queue include the **min-max heap** [16] and **deap** [37]. The textbook of Horowitz et al. [87] is a good source for learning more about these data structures.

Another related data structure is the **treap** [137], a binary search tree that includes a priority for each element, and enforces the additional property that the priority of a node is never larger than that of its parent. The intuition behind this data structure is that a binary search tree constructed from randomly selected elements tends to be balanced. By selecting the priority of each element uniformly at random, and then performing rotations to preserve the property that the priority of an element is no larger than that of its parent, it can be proven that the expected height of the search tree is $\Theta(\log n)$. The name treap is a combination of the words “tree” and “heap.”

Chapter 25

Binary Heap Data Structure

package `collection.priority`

`AbstractCollection<E>` implements *Collection<E>*

↳ `BinaryHeap<E>` implements *PriorityQueue<E>*, *Tracked<E>*

Uses: `TrackedArray` (Chapter 14)

Used By: Heap sort (Sections 11.4.3 and 15.5.3), `TaggedBinaryHeap` (Section 49.8.2)

Strengths: The binary heap makes efficient usage of space, especially if the number of elements to be held in the priority queue is known when the heap is allocated. The binary heap offers locators that track elements regardless of what mutations are made.

Weaknesses: As with all priority queue data structures, to find an element (without the use of a locator) takes linear time. Thus applications that use a priority queue will generally maintain a tracker for each element in the collection. For a binary heap, it takes logarithmic time to insert an element and to change the priority of an element. Also merging two binary heaps takes linear time.

Critical Mutators: *add*, *addAll*, *addTracked*, *clear*, *extractMax*, *remove*, *retainAll* (Note that *add*, *addAll*, and *addTracked* are not critical when the new elements have equal or lower priority than existing elements. Similarly, *remove* is not critical when a lowest priority element is removed.

Competing Data Structures: If merge must be efficient, then a different priority queue data structure should be considered. A leftist heap (Chapter 26) can perform a merge in logarithmic time, and all other methods have the same asymptotic time complexity as the binary heap. If merge must be supported and good amortized complexity is acceptable for the application, then a pairing heap (Chapter 27) is a good option. Finally, if a significant fraction of the method calls are updates to increase the priority of an element through a locator, then a Fibonacci heap (Chapter 28) should be considered.

25.1 Internal Representation

A binary heap is most naturally viewed as a perfectly balanced binary tree in which the lowest level is filled from the left. [Figure 25.1](#) shows the tree structure a binary heap for n having values 3, 4, 5, 6, 7, and 8. The other most significant property is that the priority of each node is at least as large as that of its children. While one could implement a binary heap as a pointer-based structure composed of nodes that each have a left child, right child, and parent reference (like the binary search tree seen in Chapter 32), the very restricted structure of the binary heap enables a more space

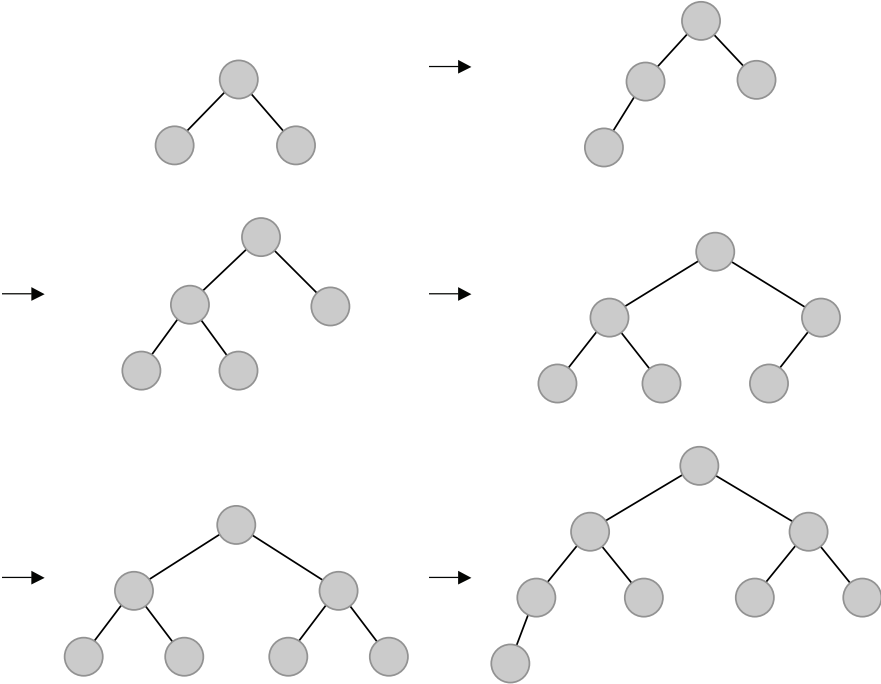


Figure 25.1
This figure illustrates the structure of binary heaps with n equal to 3, 4, 5, 6, 7, and 8, respectively.

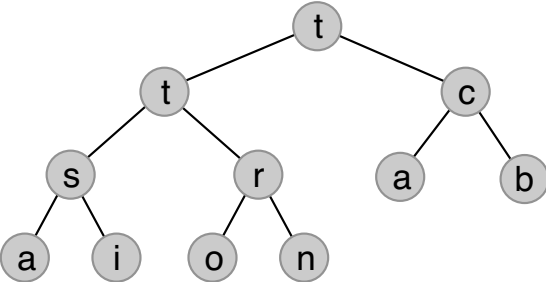


Figure 25.2
A populated example for a binary heap holding the letters of “abstraction” inserted in that order.

efficient array-based solution. Specifically, the binary heap is stored in an array where the root is stored at index 0, the next level (from left to right) is held at indices 1 and 2, and so on. Given the index of a node, the index of its parent and children can be computed using a simple arithmetic expression as described in Section 25.3.3.

Instance Variables and Constants: A binary heap wraps a positional collection *bh* that holds the elements. The constant *NOT_FOUND* is used as a return value by several internal methods when a target element is not in the collection. The boolean *tracked* is *true* if and only if trackers are being supported, as specified in the call to the constructor.

```
final int NOT_FOUND = -1;
PositionalCollection<E> bh;
boolean tracked;
```

Populated Example: Figure 25.2 shows a populated example of a binary heap holding the letters of “abstraction” inserted in the order they appear in the word.

Terminology: In addition to the definitions given in Chapter 7 and Chapter 24, we use the following definition.

- We refer to the node referenced by *bh.get(i)* as node *i*.

Abstraction Function: Let *BH* be a binary heap. The abstraction function

$$AF(BH) = \{u_0, u_1, \dots, u_{size-1}\} \text{ such that } u_p = bh.get(p).$$

That is, *AF(BH)* is the set of elements referenced by an element in *bh*.

Design Notes: The binary heap provides an example of using a wrapper as a way to create a facade for the wrapped object by both hiding methods of the underlying collection that directly refer to the position of an element, and introducing new methods that relate to the relative ordering of the elements (e.g., *max* or *extractMax*). The advantage of wrapping a positional collection is that automatic resizing and tracking are supported when needed by the binary heap.

Our binary heap implements the *Tracked* interface, but tracking adds some overhead. Therefore, the application program has the option of providing an argument to the constructor to indicate whether or not tracking support is desired. The *addTracked* method throws an unsupported operation exception when tracking is not being supported. An alternative would be to implement both a *BinaryHeap* and *TrackedBinaryHeap* as separate classes.

Optimizations: If the initial capacity provided to the constructor is guaranteed to be sufficient, then a slight efficiency improvement would be realized by using an array instead of a dynamic array. Similarly, for the tracked version, a variation of tracked array that wraps an array instead of a dynamic array could be used.

In the worst-case, it takes linear time to locate a target element *e* in the binary heap. Our implementation iterates through the collection, comparing all elements in the underlying array to *e*, stopping if an equivalent element is found. If desired, the search could be optimized to search only in subtrees for which *e* has lower priority than the subtree root. However, if *e* has a lower priority than all elements in the collection, every node would still be compared to *e*. As an illustration, the leftist heap *find* method applies this optimization strategy (Chapter 26).

25.2 Representation Properties

We inherit `SIZE` from the `AbstractCollection` class, and add the following property.

HEAPOORDERED: The priority of each node is as great as that of its descendants.

25.3 Methods

In this section we present the internal and public methods for the `BinaryHeap` class.

25.3.1 Constructors

The constructor takes *initialCapacity*, the desired initial capacity, *comp*, a user-provided comparator that defines the ordering among the elements, and *tracked*, to indicate whether or not tracking should be supported. It creates an empty priority queue with the specified characteristics.

When a tracked implementation is desired, a tracked array is wrapped. Otherwise, since the binary heap methods add and remove elements only from the back of the array, a dynamic array is wrapped.

```
public BinaryHeap(int initialCapacity, Comparator<? super E> comp, boolean tracked){
    super(comp);
    this.tracked = tracked;
    if (tracked)
        bh = new TrackedArray<E>(initialCapacity, comp);
    else
        bh = new DynamicArray<E>(initialCapacity, comp);
}
```

Several additional constructors are provided to replace some parameters by default values. The constructor that takes no parameters creates an empty priority queue that uses the default comparator.

```
public BinaryHeap(){
    this(DEFAULT_CAPACITY, DEFAULT_COMPARATOR, false);
}
```

The constructor that takes *comp*, a user-provided comparator, creates an untracked empty priority queue that orders the elements using the provided comparator, and has a default initial capacity.

```
public BinaryHeap(Comparator<? super E> comp) {
    this(DEFAULT_CAPACITY, comp, false);
}
```

The constructor that takes *initialCapacity*, the desired initial capacity, creates an empty untracked empty priority queue that uses the default comparator, and the provided initial capacity.

```
public BinaryHeap(int initialCapacity) {
    this(initialCapacity, DEFAULT_COMPARATOR, false);
}
```

The constructor that takes *initialCapacity*, the desired initial capacity and *comp*, a user-provided comparator, creates an untracked empty priority queue that uses the provided initial capacity and comparator.

```
public BinaryHeap(int initialCapacity, Comparator<? super E> comp) {
    this(initialCapacity, comp, false);
}
```

The last constructor takes *tracked*, to indicate whether tracking is desired. It creates a tracked or untracked empty priority queue depending on whether *tracked* is *true* or *false*. The priority queue is created with the default comparator, and has a default initial capacity.

```
public BinaryHeap(boolean tracked) {
    this(DEFAULT_CAPACITY, DEFAULT_COMPARATOR, true);
}
```

25.3.2 Trivial Accessors

The *getSize* and *isEmpty* methods are delegated to the wrapped array.

```
public int getSize() {return bh.getSize();}

public boolean isEmpty() {return bh.isEmpty();}
```

25.3.3 Representation Accessors

We define several representation accessors to compute the index of a node's parent, left child and right child. The *parent* method takes *i*, the index of a node, and returns the index of its parent. As long as $i \geq 1$, the parent method returns a non-negative integer. A negative return value indicates that *i* is 0, meaning that it is the root node and has no parent.

```
final int parent(int i) {return ((int) (i - 1) / 2);}
```

The *left* method takes *i*, the index of the current node, and returns the index of its left child. A return value greater than *size* - 1 indicates that no left child exists.

```
final int left(int i) {return 2 * i + 1;}
```

The *right* method takes *i*, the index of the current node, and returns the index of its right child. A return value that is greater than *size* - 1 indicates that no right child exists.

```
final int right(int i) {return 2 * i + 2;}
```

25.3.4 Algorithmic Accessors

The *max* method returns a highest priority element. It throws a *NoSuchElementException* when this collection is empty.

```

public E max() {
    if (isEmpty())
        throw new NoSuchElementException();
    return bh.get(0);
}

```

Correctness Highlights: Follows directly from `HEAPORDERED` since all descendants of the root have an equal or lower priority. ■

Although there is no efficient search method for a binary heap, we include a linear time internal method to locate an element. The *find* method takes *element*, the target, and returns a reference to a heap node holding an equivalent element, or *null* if there is no equivalent element in this collection.

```

protected int find(E element){
    for (int i = 0; i < bh.getSize(); i++)
        if (equivalent(element, bh.get(i)))
            return i;
    return NOT_FOUND;
}

```

The *contains* method takes *element*, the target, and returns *true* if and only if an equivalent element exists in this collection.

```

public boolean contains(E element) {
    return bh.contains(element);
}

```

The method *getEquivalentElement* takes *element*, the target, and returns an equivalent element from the collection. It throws a *NoSuchElementException* when there is no equivalent element in this collection.

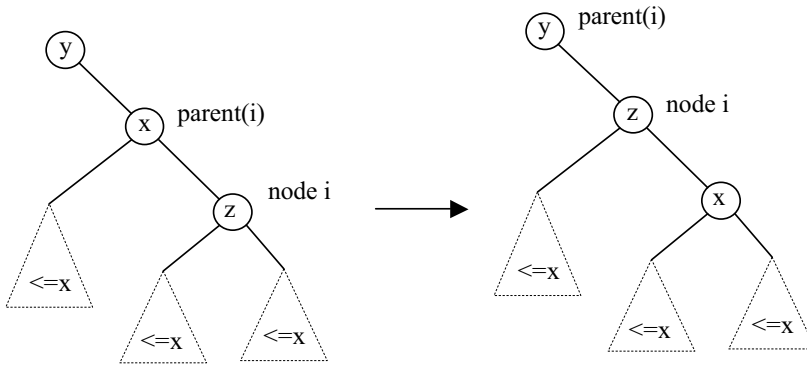
```

public E getEquivalentElement(E target) {
    int index = find(target);
    if (index == NOT_FOUND)
        throw new NoSuchElementException();
    else
        return (E) bh.get(index);
}

```

25.3.5 Representation Mutators

Increasing or decreasing the priority of an element can violate the property `HEAPORDERED`. There are two internal methods that are used to restore `HEAPORDERED` in such cases. One way to think about the `HEAPORDERED` property is that for every leaf ℓ , the elements along the path from ℓ to the root are sorted from lowest to highest priority. When the priority of an element is increased, then the path from a leaf to the root containing this node may no longer be sorted. The *fixUpward* method repeatedly swaps this element with its parent until the elements in that path are sorted. More

**Figure 25.3**

An illustration of one iteration of the while loop of *fixUpward*. By **HEAPORDERED**, in the heap drawn on the left, all elements in the subtrees shown as triangles have priority no larger than x . Since x and z are only swapped when $z \geq x$, it follows that z has priority is at least as large as all of its descendants, and likewise x has priority at least as large as all of its descendants. So the only possible violation continues to be between z (node i) and its parent.

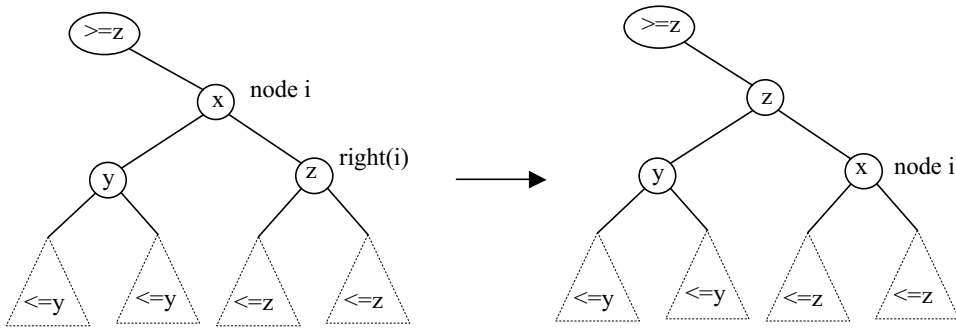
specifically, the *fixUpward* method takes i , the index of the node causing a possible violation with its parent, and restores **HEAPORDERED** by repeatedly swapping the violating node with its parent until **HEAPORDERED** is restored. This method requires that the only violation to **HEAPORDERED** occurs between node i and its parent.

```
private void fixUpward(int i) {
    while ((i > 0) && comp.compare(bh.get(i), bh.get(parent(i))) > 0) { //parent(i) > i
        bh.swap(i, parent(i));
        i = parent(i);
    }
}
```

Correctness Highlights: We argue inductively that after each iteration of the while loop, the only possibly violation of **HEAPORDERED** occurs between node i and its parent. The only change made at each iteration is to swap node i with its parent when $a.get(i) > a.get(parent(i))$. By the inductive hypothesis, it follows that all other descendants of $parent(i)$ have a priority no more than that of $parent(i)$. (See Figure 25.3.) Let i' be the value of i after the swap. The only possible violation that remains is between $i' = parent(i)$ and $parent(i')$. This method continues until $i = 0$ which indicates that it is the root, or the parent of node i has a higher priority at which time **HEAPORDERED** holds for all nodes. Thus **HEAPORDERED** is restored upon termination. ■

When the priority of an element is decreased, then it must be moved down the tree until every path from a leaf to the root containing this node is in sorted order. The *fixDownward* method, which is often called **heapify**, repeatedly swaps the violating node with its higher priority child until **HEAPORDERED** is restored. More specifically, The *fixDownward* method takes i , the index of the node causing a possible violation with its children, and restores **HEAPORDERED**. This method requires that the descendants of node i , if any, satisfy the **HEAPORDERED** property.

```
private void fixDownward(int i) {
    int max = i; //index of higher priority child
    while (true) {
```


**Figure 25.4**

An illustration of one iteration of the while loop of the *fixDownward* method. By **HEAPORDERED**, in the heap drawn on the left, all descendants of *y* have priority at most that of *y*, and all descendants of *z* have priority at most that of *z*. When a swap occurs, it is between the higher priority child of *x* which has priority larger than *x*. (In the illustration *z* is such a child.) By **HEAPORDERED** it follows that all ancestors of *x* must have priority at least as large as that of *z* since by the inductive hypothesis the only violation of **HEAPORDERED** is between *x* and its children. The drawing on the right shows the heap after *x* and *z* are interchanged. Observe that the only possible violation of **HEAPORDERED** occurs between *x* (node *i*) and its children.

```

int left = left(i);
int right = left + 1;
if (right > bh.getSize())                               //node i is a leaf, so has no children
    return;
else if (right == bh.getSize()
        || comp.compare(bh.get(left), bh.get(right)) > -1) //node i only has left child
    max = left;
else
    max = right;
if (comp.compare(bh.get(i), bh.get(max)) < 0) {           //max has higher priority
    bh.swap(i, max);
    i = max;
}
else                                                       //node i has a priority at least as high as its children
    return;
}
}

```

Correctness Highlights: We argue inductively that after each iteration of the while loop, the only possible violation of **HEAPORDERED** occurs between node *i* and one of its children. Once node *i* has no children with a higher priority, **HEAPORDERED** had been restored. Consider when one of node *i*'s children has a higher priority than it. In this case node *max* has the higher priority of the three, and it is swapped with node *i*. By **HEAPORDERED** and the inductive hypothesis, after this swap is made the only possible violation of **HEAPORDERED** that remains is between node *i* (in its new location) and its children. (See Figure 25.4.) Thus upon termination **HEAPORDERED** is restored. ■

25.3.6 Content Mutators

A priority queue supports only the update of an element through a locator, which is performed using the internal *update* method that takes *p*, the position of the element to update, and *element*, the replacement element. It requires that $0 \leq p < \text{size}$.

```
protected void update(int p, E element){
    bh.set(p, element);                                //update the element at position p
    if (comp.compare(element, bh.get(parent(p))) > 0)
        fixUpward(p);
    else
        fixDownward(p);
}
```

Correctness Highlights: Let *e* be the element at position *p*. When the priority for *e* does not change, clearly HEAPOORDERED is preserved. We consider the cases in which the priority of *e* increases or decreases.

value > e: The only possible violation of HEAPOORDERED is between the element at position *p* and its parent. Thus, the requirement for *fixUpward* is satisfied. By the correctness of *fixUpward*, HEAPOORDERED is restored.

value < e: The only possible violation of HEAPOORDERED is between the element at position *p* and its children. Thus, the requirement for *fixDownward* is satisfied. By the correctness of *fixDownward*, HEAPOORDERED is restored.

The rest of the correctness follows from the positional collection *get* and *set* methods. ■

Methods to Perform Insertion

The *add* method takes *element*, the element to add. The new element is always placed in the first open leaf position, and then HEAPOORDERED is restored using *fixUpward*. A sequence of insertions is illustrated in [Figure 25.5](#).

```
public void add(E element) {
    bh.addLast(element);                                //adds element and updates size
    fixUpward(bh.getSize()-1);                          //new element at position size-1
}
```

Correctness Highlights: The new element is added to the end of the positional collection that maintains the structure of the heap. Observe that the only possible violation of HEAPOORDERED is between the new element and its parent, which satisfies the conditions on *fixUpward*. The rest of the correctness follows from SIZE, the binary heap *addLast* method, and *fixUpward*. ■

The *addTracked* method takes *element*, the element to add. It returns a tracker to the newly added element. It throws an *UnsupportedOperationException* when tracking is not being supported, as specified in the constructor call.

```
public PriorityQueueLocator<E> addTracked(E element) {
    if (!tracked)
        throw new UnsupportedOperationException();
    PositionalCollectionLocator<E> t = ((TrackedArray<E>) bh).addTracked(element);
}
```

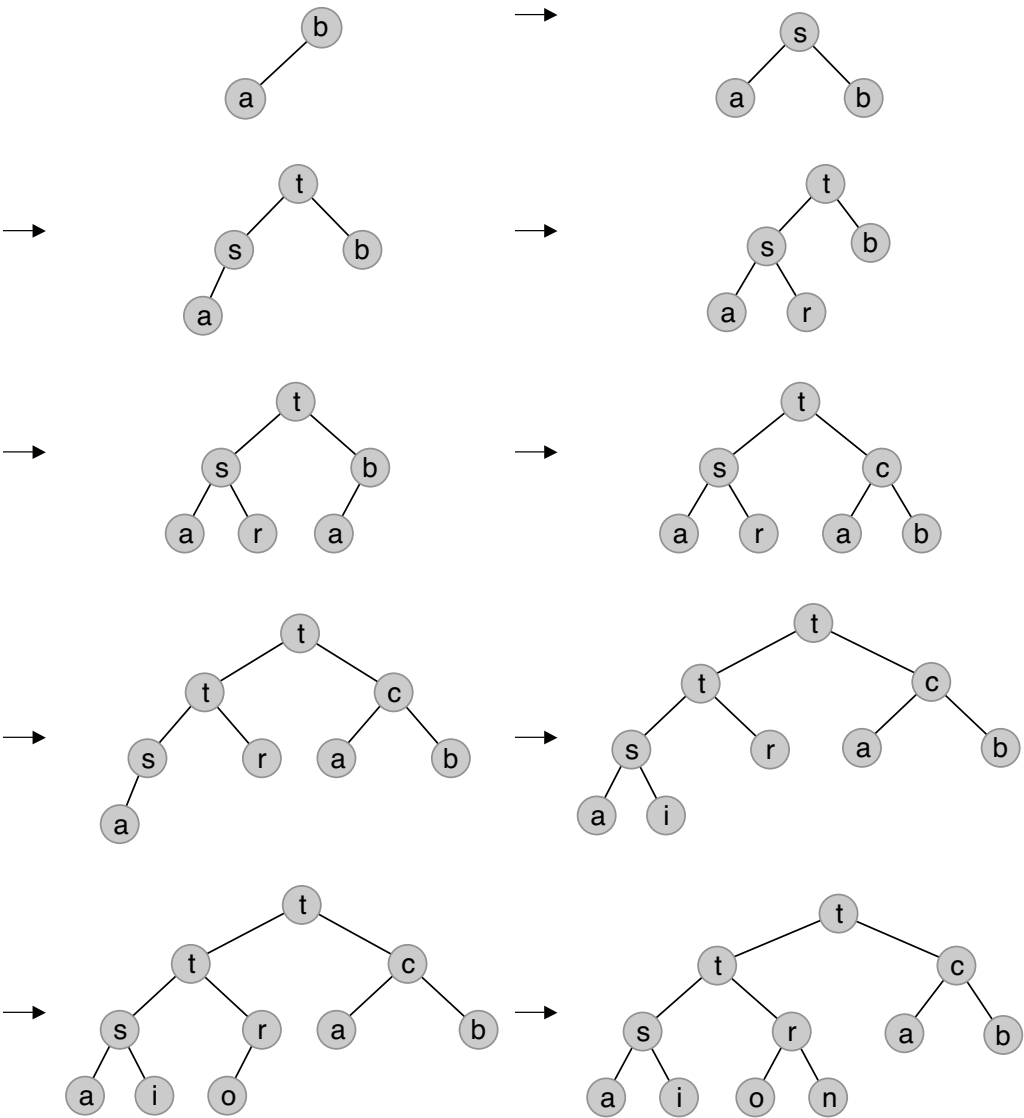
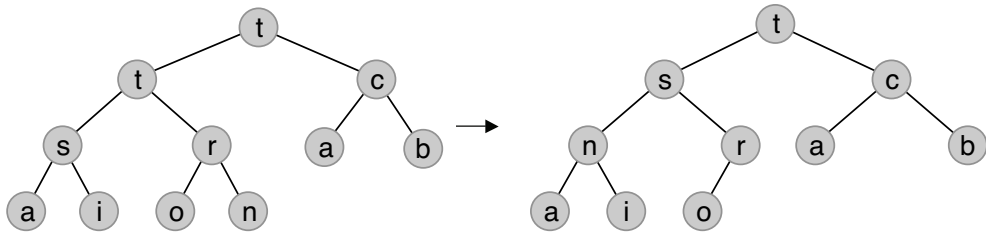


Figure 25.5
Binary heap insertion example using the letters in “abstraction” inserted in that order. Each insertion adds the new element at the first open leaf position (not shown), and then repeatedly swaps the inserted node with its parent until HEAPOORDERED is restored.

**Figure 25.6**

Binary heap *extractMax* example.

```

    fixUpward(bh.getSize()-1);
    return new BinaryHeapLocator(t);
}

```

Correctness Highlights: Like the *add* method, except that a tracker is created and returned. ■

Recall that *addAll* takes *c*, the collection to be added to this collection. While one could iterate through *c*, and insert each element into the binary heap, a more efficient implementation is to add new elements to the end of *bh*, potentially creating many violations of *HEAPORDERED*. Then *fixDownward* is applied to each non-leaf node from the bottom of the binary heap towards the root.

```

public void addAll(Collection<? extends E> c) {
    bh.addAll(c);
    for (int i = (bh.getSize()/2 - 1); i ≥ 0; i--)
        fixDownward(i);
}

```

Correctness Highlights: We first prove that this method has the invariant that when *fixDownward* is called for node *i*, all $j > i$, $T(j)$ satisfies *HEAPORDERED*. We argue by induction on *i* from *size* - 1 down to 0. When node *i* is a leaf, this invariant trivially holds, since all nodes $j > i$ are also leaves and have no children to violate *HEAPORDERED*. Consider when node *i* is not a leaf. By the invariant, the heaps rooted at both of its children satisfy *HEAPORDERED*. Thus, the only possible violation in $T(i)$ is between *i* and its children. Thus the requirement of *fixDownward* is satisfied, which implies that after *fixDownward* executes, $T(i)$ will satisfy *HEAPORDERED*. This completes the proof of the stated invariant.

By this invariant, once the for loop has terminated with $i = 0$, the entire binary heap satisfies *HEAPORDERED*. *SIZE* is preserved by the positional collection *addAll* method. ■

Methods to Perform Deletion

The *extractMax* method removes and returns the highest priority element from the collection. It throws a *NoSuchElementException* when called on an empty collection.

```

public E extractMax() {
    bh.swap(0, bh.getSize()-1);           //swap root with last element
    E data = bh.removeLast();             //remove max priority element
}

```

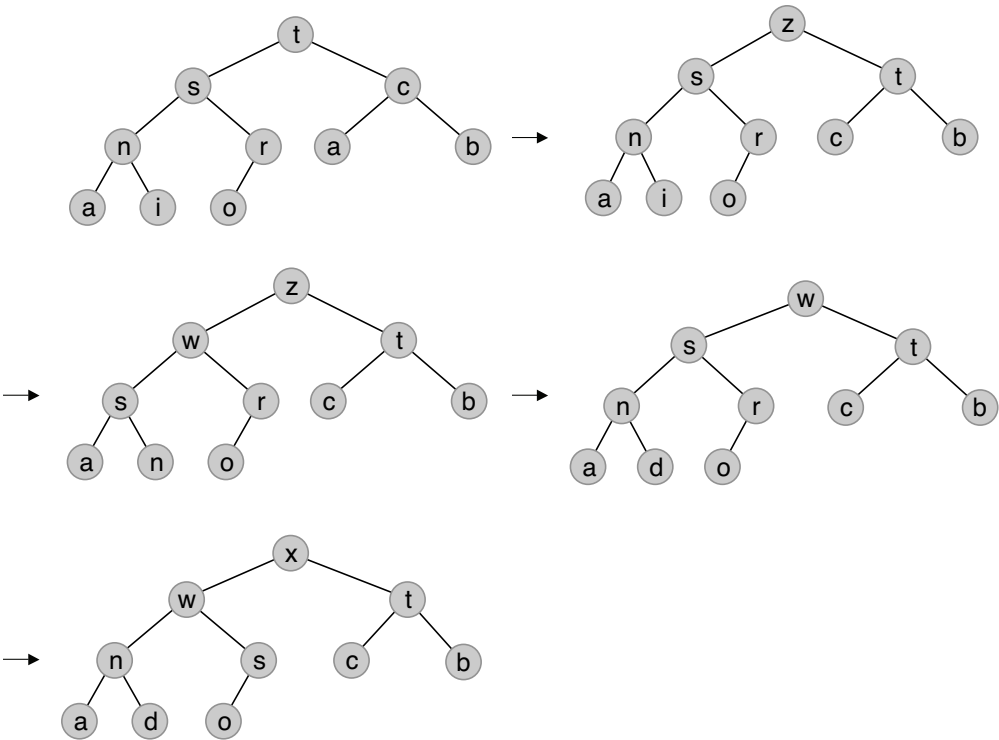


Figure 25.7
Binary heap example of updating the values (using locators). First one of the “a”s is changed to “z.” Next “i” is changed to “w.” Then the “z” is changed to a “d,” and finally one of the “t”s is changed to an “x.”

```

        fixDownward(0);                                //restore HeapOrdered property
        return data;                                    //return max priority element
    }

```

Correctness Highlights: By `HEAPORDERED`, the element at position 0 has the maximum priority. By `SIZE`, the element to be removed is exchanged with the last element, which is then removed and stored in *data*. Since `HEAPORDERED` held prior to the execution of this method, when *fixDownward* is called, the left and right subtree of the root both satisfy `HEAPORDERED`. Thus the only possible violation of `HEAPORDERED` is between the root and its children, which satisfies the requirement for *fixDownward*(0). So *fixDownward* restores `HEAPORDERED`. The rest of the correctness follows from that of the positional collection *swap* and *removeLast* methods. ■

The internal *remove* method that takes *p*, the position of the element to remove. It requires that $0 \leq p < \text{size}$.

```

protected void remove(int p) {
    E removed = bh.get(p);
    int lastPos = getSize()-1;
    if (p == lastPos)
        bh.removeLast();
    else {
        bh.swap(lastPos, p);                                //move element to remove to last position
        bh.removeLast();                                    //and remove it
        if (comp.compare(bh.get(p), removed) ≥ 0)           //priority at pos p least as high
            fixUpward(p);
        else                                                 //priority at pos p decreases
            fixDownward(p);
        version.increment();                                //invalidate all locators
    }
}

```

Correctness Highlights: The *remove* method is very similar to the *extractMax* method. By the correctness of the positional collection *get* method, the element to be removed is stored in *removed*.

For the sake of efficiency (and also to invalidate the locators only when necessary), we treat the removal of the last element in the binary heap as a special case since it does not require the heap to be restructured. We now consider when $p \neq \text{lastPos}$. Observe that the element to be removed is swapped with the last element in the binary heap, and then removed. But now the element in position *p* might violate `HEAPORDERED` with respect to either its children or parent.

If the priority of the element now in position *p* is equal or larger than the priority of the removed element, then the only possible violation of `HEAPORDERED` is between node *p* and its parent. Thus, the requirement for *fixUpward* is satisfied, and by the correctness of *fixUpward* `HEAPORDERED` is restored.

The only other possibility is that the priority of the element now in position *p* is smaller than the priority of the removed element. In this case, the only possible violation of `HEAPORDERED` is between node *p* and its children. Thus, the requirement for *fixDownward* is satisfied. By the correctness of *fixDownward*, `HEAPORDERED` is restored.

The rest of the correctness follows from that of *getSize*, *swap*, and *removeLast*. ■

The public *remove* method takes *target*, the element to remove. It removes from the collection an arbitrary element (if any) equivalent to the target. It returns *true* if an element was removed, and *false* otherwise. Observe that this method uses the linear time *find* method. To locate the target element efficiently, a tracker is required.

```
public boolean remove(E target) {
    int p = find(target);
    if (p == NOT_FOUND)
        return false;
    else {
        remove(p);
        return true;
    }
}
```

//find index of equivalent value

The *clear* method removes all elements from the collection.

```
public void clear(){
    bh.clear();
}
```

The *retainAll* method takes *c*, a collection, and updates the current collection to contain only elements that are also in *c*. In other words, the current collection is updated to be the intersection between itself and *c*.

```
public void retainAll(Collection<E> c) {
    bh.retainAll(c);
    for (int i = (bh.getSize()/2 - 1); i ≥ 0; i--)
        fixDownward(i);
}
```

Correctness Highlights: Like the correctness argument for *addAll*, except that it instead relies on the correctness of the positional collection *retainAll* method. ■

25.3.7 Locator Initializers

The *iterator* method creates a new locator that is at FORE.

```
public PriorityQueueLocator<E> iterator() {
    return new BinaryHeapLocator(bh.iterator());
}
```

The method *getLocator* takes *element*, the target. It returns a locator positioned at the given element. It throws a *NoSuchElementException* when there is no equivalent element in this collection. This method runs in worst-case linear time.

```
public PriorityQueueLocator<E> getLocator(E element){
    return new BinaryHeapLocator(
        (PositionalCollectionLocator<E>) bh.getLocator(element));
}
```

25.4 Locator Inner Class

`AbstractCollection<E>.AbstractLocator<E>` implements *Locator<E>*

↳ **BinaryHeapLocator** implements *PositionalCollectionLocator<E>*

The binary heap locator wraps the positional collection locator. One advantage of this approach as compared to directly using the positional collection locator, is that methods such as *set* that are not appropriate for a binary heap locator are not exposed. In addition, the *update* method must be supported. All of the methods except *update* are delegated to the positional collection locator.

```
PositionalCollectionLocator<E> loc; //wrapped locator
```

The constructor takes *loc*, a reference to the node to place the locator.

```
protected BinaryHeapLocator(PositionalCollectionLocator<E> loc){
    this.loc = loc;
}
```

Recall that *inCollection* returns *true* if and only if the element at the locator is in the collection.

```
public boolean inCollection() {
    return loc.inCollection();
}
```

The *get* method returns the element at the locator position. It throws a *NoSuchElementException* when locator is not at an element in the collection.

```
public E get() {
    return loc.get();
}
```

The *advance* method moves the locator to the next element in the iteration order (or AFT if it is currently at the last element). It returns *true* if and only if after the update, the locator is at an element of the collection. It throws an *AtBoundaryException* when the locator is at AFT since there is no place to advance.

```
public boolean advance() {
    return loc.advance();
}
```

The *retreat* method moves the locator to the previous element in the iteration order (or FORE if it is currently at the first element). It returns *true* if and only if after the update, the locator is at an element of the collection. It throws an *AtBoundaryException* when the locator is at FORE since then there is no place to retreat.

```
public boolean retreat() {
    return loc.retreat();
}
```

The *hasNext* method that returns *true* if and only if advancing it would leave the locator at an element in the collection.

```
public boolean hasNext() {
    return loc.hasNext();
}
```


The *update* method takes *element*, the new element to replace the one at the locator position.

```
public void update(E element) {
    int p = loc.getCurrentPosition();
    BinaryHeap.this.update(p, element);
}
```

The *increasePriority* method replaces the tracked element by *element*. It requires that the given parameter is greater than *e*, or that *e* is the parameter being passed and its value has been mutated to have a higher priority than it had previously. That is, it is acceptable practice to mutate the element to have a higher priority and then immediately call *increasePriority* to restore the properties of the priority queue.

```
public void increasePriority(E element) {
    int p = loc.getCurrentPosition();
    BinaryHeap.this.bh.set(p, element);
    BinaryHeap.this.fixUpward(p);
}
```

The *decreasePriority* method replaces the tracked element by *element*. It requires that the given parameter is less than *e*, or that *e* is the parameter being passed and its value has been mutated to have a lower priority than it had previously. That is, it is acceptable practice to mutate the element to have a lower priority and then immediately call *decreasePriority* to restore the properties of the priority queue.

```
public void decreasePriority(E element) {
    int p = loc.getCurrentPosition();
    BinaryHeap.this.bh.set(p, element);
    BinaryHeap.this.fixDownward(p);
}
```

The *remove* method removes the element at the locator position, and updates the locator to be at the element in the iteration order that preceded the one removed. It throws a *NoSuchElementException* when called on a locator at FORE or AFT.

```
public void remove() {
    int p = loc.getCurrentPosition();
    BinaryHeap.this.remove(p);
}
```

25.5 Performance Analysis

The asymptotic time complexities of all public methods for the *BinaryHeap* class are shown in [Table 25.8](#), and the asymptotic time complexities for all of the public methods of the *BinaryHeapLocator* class are given in [Table 25.9](#).

The space complexity for a binary heap is just that of a dynamic array (if the heap is untracked) or a tracked array (if the heap is tracked) of size n . For an untracked implementation where the constructor provides a desired capacity of n , only n references are needed.

We now discuss the time complexity analysis. Since a binary heap is a perfectly balanced binary tree, the height is at most $\lfloor \log_2 n \rfloor$ where there are n elements in the heap. Since *fixUpward* takes

method	time complexity
<i>max()</i>	$O(1)$
<i>add(o)</i> <i>addTracked(o)</i> <i>extractMax()</i>	$O(\log n)$ $O(\log n)$ $O(\log n)$
<i>addAll(c)</i> <i>addAll(BinaryHeap c)</i> <i>clear()</i> <i>contains(o)</i> <i>getEquivalentElement(o)</i> <i>getLocator(o)</i> <i>remove(o)</i> <i>toArray()</i> <i>toString()</i> <i>trimToSize()</i> <i>accept()</i>	$O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$
<i>ensureCapacity(x)</i>	$O(x)$
<i>retainAll(c)</i>	$O(c n)$

Table 25.8 Summary of the asymptotic time complexities for the *PriorityQueue* public methods when implemented using a binary heap.

locator method	complexity
<i>advance()</i> <i>get()</i> <i>hasNext()</i> <i>next()</i> <i>retreat()</i>	$O(1)$ $O(1)$ $O(1)$ $O(1)$ $O(1)$
<i>remove()</i> <i>update(o), increase priority</i> <i>update(o), decrease priority</i>	$O(\log n)$ $O(\log n)$ $O(\log n)$

Table 25.9 Summary of the time complexities for the binary heap locator methods.

a single pass up the tree with just one comparison made at each level, it has worst-case logarithmic cost. Likewise, *fixDownward* takes a single pass down the tree. Since each node has at most two children, constant time is spent at each level, yielding a worst-case logarithmic cost.

Since the highest priority element is always at the root (e.g., at index 0 of the positional collection), it can be found in constant time. To find an arbitrary element takes worst-case linear time since the target might be compared to all elements in the collection. For *insert*, the dominant cost is that of *fixUpward*, which is logarithmic. Similarly, once an element has been located in the binary heap, the time to remove it is dominated by the cost of *fixDownward*. So *extractMax* has logarithmic cost. To update the priority of an element, either *fixUpward* or *fixDownward* is the dominant cost, so updating a priority has logarithmic cost.

We now argue that the worst-case time complexity of *addAll* is linear. This method and its analysis are due to Floyd [55]. The time to insert all n elements into the array is linear. We define the height of node x to be the height of the subtree rooted at x . Observe that there are at most $\lceil n/2^{h+1} \rceil$ nodes at height h . We also use the fact that $\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$. The time complexity of *addAll* is dominated by the calls to the *fixDownward* method which is the height of the subtree. Adding up those costs yield a worst-case time complexity asymptotically upper bounded by

$$\sum_{h=0}^{\lceil \log_2 n \rceil} h \cdot \left\lceil \frac{n}{2^{h+1}} \right\rceil = O\left(\frac{n}{2} \cdot \sum_{h=0}^{\log_2 n} \frac{h}{2^h}\right) = O(n).$$

Since a binary heap is built upon an array, *ensureCapacity(x)* requires allocating a new array of size x , and initializing all elements. Then all n elements must be copied into the new array. So the cost is $O(n + x)$. Similarly, *trimToSize* must allocate a new array of size n and copy all n elements into it yielding an $O(n)$ cost.

The *clear*, *toString*, *toArray*, and *accept* methods just traverse the array, spending constant time at each element. Thus the overall cost is $O(n)$.

The *retainAll* method performs a search for each element in the provided collection c . Any element not found in c is removed. In the worst-case, a linear time search must be performed for each element. This is the dominant cost, since removing an element once located takes only logarithmic time.

Finally, we analyze the time complexity of the locator methods. Since the iteration order just uses the order of the elements in the underlying array, *advance*, *retreat*, *next*, and *hasNext* take constant time. Finally, since the index for a locator's element is known, the cost of calling *remove* on a locator is dominated by either a call to *fixUpward* or *fixDownward*, both of which take logarithmic time.

25.6 Quick Method Reference

BinaryHeap Public Methods

- p. 356 *BinaryHeap*()
- p. 356 *BinaryHeap*(Comparator<? super E> comp)
- p. 357 *BinaryHeap*(boolean tracked)
- p. 356 *BinaryHeap*(int initialCapacity)
- p. 357 *BinaryHeap*(int initialCapacity, Comparator<? super E> comp)
- p. 356 *BinaryHeap*(int initialCapacity, Comparator<? super E> comp, boolean tracked)
- p. 98 void **accept**(Visitor<? super E> v)
- p. 361 void **add**(E element)
- p. 100 void **addAll**(Collection<? extends E> c)

- p. 361 PriorityQueueLocator<E> **addTracked**(E element)
- p. 100 void **clear**()
- p. 97 boolean **contains**(E value)
- p. 99 void **ensureCapacity**(int capacity)
- p. 363 E **extractMax**()
- p. 96 int **getCapacity**()
- p. 96 Comparator<? super E> **getComparator**()
- p. 98 E **getEquivalentElement** (E target)
- p. 366 PriorityQueueLocator<E> **getLocator**(E element)
- p. 96 int **getSize**()
- p. 96 boolean **isEmpty**()
- p. 366 PriorityQueueLocator<E> **iterator**()
- p. 357 E **max**()
- p. 366 boolean **remove**(E target)
- p. 100 void **retainAll**(Collection<E> c)
- p. 97 Object[] **toArray**()
- p. 97 E[] **toArray**(E[] array)
- p. 98 String **toString**()
- p. 99 void **trimToSize**()

BinaryHeap Internal Methods

- p. 97 int **compare**(E e1, E e2)
- p. 97 boolean **equivalent**(E e1, E e2)
- p. 358 int **find**(E element)
- p. 359 void **fixDownward**(int i)
- p. 359 void **fixUpward**(int i)
- p. 357 int **left**(int i)
- p. 357 int **parent**(int i)
- p. 365 void **remove**(int p)
- p. 357 int **right**(int i)
- p. 99 void **traverseForVisitor**(Visitor<? super E> v)
- p. 361 void **update**(int p, E element)
- p. 98 void **writeElements**(StringBuilder s)

BinaryHeap.BinaryHeapLocator Public Methods

- p. 367 boolean **advance**()
- p. 368 void **decreasePriority**(E element)
- p. 367 E **get**()
- p. 367 boolean **hasNext**()
- p. 101 void **ignoreConcurrentModifications**(boolean ignore)
- p. 101 void **ignorePriorConcurrentModifications**()
- p. 367 boolean **inCollection**()
- p. 368 void **increasePriority**(E element)
- p. 101 E **next**()
- p. 368 void **remove**()
- p. 367 boolean **retreat**()
- p. 368 void **update**(E element)

BinaryHeap.BinaryHeapLocator Internal Methods

- p. 367 *BinaryHeapLocator*(PositionalCollectionLocator<E> loc)
- p. 101 void **checkValidity**()
- p. 101 void **updateVersion**()

Chapter 26

Leftist Heap Data Structure

package `collection.priority`

`AbstractCollection<E>` implements *Collection<E>*

↳ `LeftistHeap<E>` implements *PriorityQueue<E>*, *Tracked<E>*

Uses: Java references

Used By: `TaggedLeftistHeap` (Section 49.8.3)

Strengths: The leftist heap supports a logarithmic time operation to merge two heaps. All other methods have the same asymptotic time complexities as the binary heap.

Weaknesses: Space usage is significantly higher than for the binary heap, especially if it is threaded to support constant-time iteration.

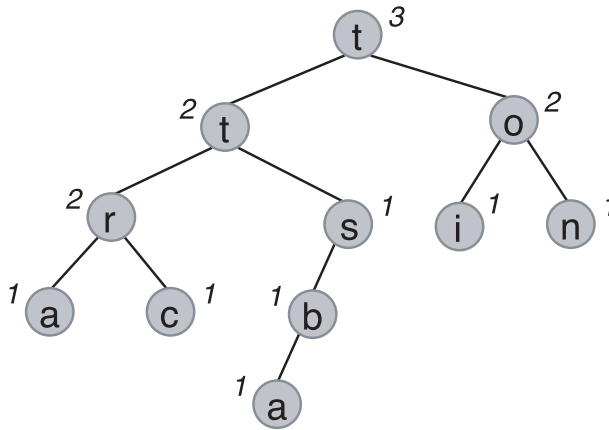
Critical Mutators: In an unthreaded implementation *add*, *addAll*, *addTracked*, *extractMax*, *remove*, and *retainAll* are critical mutators. In a threaded implementation there would be no critical mutators.

Competing Data Structures: If space efficiency is important and an efficient merge is not needed, then a binary heap (Chapter 25) should be considered. If merge must be supported and good amortized complexity (but high actual complexity for some methods) is acceptable for the application, then a pairing heap (Chapter 27) is a good option. Finally, if a significant fraction of the methods involve increasing the priority of an element through a locator, then a Fibonacci heap (Chapter 28) should be considered.

26.1 Internal Representation

Similar to a binary heap, a leftist heap is a heap-ordered binary tree. However, the elements are not stored in an array. Instead, a separate node object is allocated for each element. For ease of exposition, we conceptually view all *null* child references as referencing an imaginary **frontier node**. Sometimes the term **external node** is used in place of frontier node. For any node *x* in a leftist heap, we define *x.dtf* (**distance to frontier**) as the shortest distance from *x* to the frontier node.

Unlike a binary heap, which is a completely balanced binary tree, the only structural property preserved by the leftist heap is $x.\text{left.dtf} \geq x.\text{right.dtf}$, which is used to bound the depth of the tree. Because of this relaxed structure, a pointer-based structure comprised of heap nodes is the most appropriate internal representation for a leftist heap. Each heap node maintains a reference to an

**Figure 26.1**

A populated example for a leftist heap holding the letters of “abstraction.” The shortest distance to a frontier node (dtf) is shown next to each node.

associated element, a reference to its left child, a reference to its right child, a reference to its parent, and its *dtf* value.

Instance Variables and Constants: The variables *size*, *comp*, *DEFAULT_CAPACITY*, and *version* are inherited from *AbstractCollection*. We introduce the constant *FRONTIER* which is set to *null*, to increase the readability of the code without actually creating the (imagined) frontier node. The variable *root* refers the root of the leftist heap.

Finally, to allow the true merging of two leftist heaps (each of which might have some tracked element and active iterators), we introduce a union-find node *proxy* that references the representative leftist heap among a set that have been merged into a single leftist heap. Section 6.8 discusses this application of the union-find data structure in depth.

```
final static LeftistHeapNode FRONTIER = null;           //used for any null child
LeftistHeapNode<E> root;                               //reference to the root of the leftist heap
UnionFindNode<LeftistHeap<E>> proxy;                  //to support of efficient union
```

Populated Example: Figure 26.1 shows a populated example of a leftist heap holding the letters of “abstraction.”

Terminology: In addition to the definitions given in Chapter 7 and Chapter 24, we introduce the following definitions.

- We define *dtf* for node *x* formally as follows:

$$dtf(x) = \begin{cases} 0 & \text{if } x \text{ is a frontier node} \\ 1 + \min\{dtf(x.left), dtf(x.right)\} & \text{otherwise.} \end{cases}$$

- The **iteration order** is defined by a preorder traversal from the root. More formally, the iteration order is given by *seq*(*root*), defined recursively as

$$seq(x) = \begin{cases} \langle \rangle, & \text{if } x \text{ is the imagined frontier node} \\ x.data + seq(x.left) + seq(x.right) & \text{otherwise.} \end{cases}$$

Abstraction Function: Let LH be a leftist heap. The abstraction function is:

$$AF(LH) = seq(root).$$

Optimizations: Space usage is minimized by only supporting iteration through a visiting iterator (Section 8.5). An alternative approach that would provide constant-time support for iteration at the cost of higher space usage is a threaded implementation, in which each heap node also includes a *next* and *prev* reference for the corresponding heap nodes in the iteration order. Our implementation of a pairing heap (Chapter 27) illustrates such an approach. A third alternative would be to structurally find the successor of a node in the depth first ordering as iteration proceeds, as illustrated in the locator of the binary search tree (Chapter 32). This intermediate approach uses little space, but the time to advance the iterator can be proportional to the depth of the tree.

To improve efficiency, we replace the *traverseForVisitor* of the *AbstractCollection* class, which uses an iterator, by a method that uses a preorder traversal of the leftist heap. Also the *find* method uses a preorder traversal with the added optimization of only recursively searching in a subtree when the desired element has a smaller priority than that at the root of the subtree.

Since all mutations that invalidate a locator are made through the internal *merge* method, the modification count is only updated in that method. However, the *merge* method is recursive and could call itself up to $O(\log n)$ times, resulting in an equal number of updates to the modification count even though a single such update would be sufficient. An alternative would be to instead update the modification count just before *merge* is called by any method. However, it would be a mistake to blindly invalidate all locators in each public critical mutator, because when *merge* is not called, there is no need to invalidate the locators.

26.2 Representation Properties

We inherit *SIZE* from the *AbstractCollection* class and add the following properties.

REACHABLE: An element is in the collection exactly when it is held within some heap node in $T(root)$.

HEAPORDERED: The priority of each reachable heap node is at least as large as that of its descendants. More formally, for all $x \in T(root) - root$, $x.parent.element \geq x.element$.

DTF: For every reachable heap node x , $x.dtf = dtf(x)$.

LEFTLEANING: For every reachable heap node x , $dtf(x.left) \geq dtf(right.dtf)$. Recall that $dtf(FRONTIER)$ is 0.

PARENT: For each reachable heap node x , either $x.child(i) = FRONTIER$ or $x.child(i).parent = x$. Finally, $root.parent = null$.

INUSE: For any heap node x , $x.left = null$ if and only if $x \notin T(root)$. That is, a heap node is in use exactly when $x.left \neq null$.

Observe that by exploiting *LEFTLEANING*, we can write a simpler equivalent definition for *dtf* as follows:

$$dtf(x) = \begin{cases} \langle \rangle, & \text{if 0 is a frontier node} \\ 1 + dtf(x.right) & \text{otherwise.} \end{cases}$$

26.3 Leftist Heap Node Inner Class

LeftistHeapNode<E>

Each leftist heap node includes a reference to the element, its left child, its right child, and its parent. In addition, to enable mutators to efficiently preserve LEFTLEANING, each node keeps track of its shortest distance to a frontier node

```
E element;                                //reference to the element
LeftistHeapNode<E> left;                  //reference to left child
LeftistHeapNode<E> right;                 //reference to right child
LeftistHeapNode<E> parent;                //reference to parent
int dtf = 1;                             //shortest distance to a frontier node
```

The constructor takes *element*, the value to be held in this node.

```
public LeftistHeapNode(E element) {
    this.element = element;
}
```

The *markDeleted* method modifies this node to indicate that its element, which may be tracked, has been removed from the collection.

```
final void markDeleted() {
    left = this;
}
```

Correctness Highlights: The maintains INUSE. ■

The *isDeleted* method returns *true* if and only if this occurrence of the element held in this node has been deleted.

```
final boolean isDeleted() {
    return left == this;
}
```

Correctness Highlights: Follows from INUSE. ■

The *dtfLeft* method returns the shortest distance to a frontier node in the left subtree.

```
int dtfLeft() {
    return (left == FRONTIER) ? 0 : left.dtf;
}
```

Correctness Highlights: By definition, if *left* is a frontier, then 0 is the correct return value. The rest of the correctness follows from DTF. ■

The *dtfRight* method returns the shortest distance to a frontier node in the right subtree.

```
int dtfRight() {
    return (right == FRONTIER) ? 0 : right.dtf;
}
```

Correctness Highlights: Like that for *dtfLeft*. ■

The *setLeft* method takes *x*, a reference to a heap node, and makes *x* the left child of this heap node.

```
void setLeft(LeftistHeapNode<E> x) {
    this.left = x;
    x.parent = this;
    swapChildrenAsNeeded();
}
```

Correctness Highlights: This method sets the parent pointer of *x* to maintain PARENT with respect to this relationship between *x* and the heap node on which this method is called. By the correctness of *swapChildrenAsNeeded*, LEFTLEANING is preserved. ■

The *setRight* method that takes *x*, a reference to a heap node, and makes *x* the right child of this heap node.

```
void setRight(LeftistHeapNode<E> x) {
    this.right = x;
    x.parent = this;
    swapChildrenAsNeeded();
}
```

Correctness Highlights: Like that for *setLeft*. ■

The *swapChildren* method interchanges the left and right children when the right child has a larger dtf. The method returns *true* when the children are swapped. It preserves LEFTLEANING and DTF for this node, but because the dtf of this node may change, these properties may be violated for this node's parent. Therefore, any caller of this method ensure that these properties are restored for the ancestors as necessary. This method requires that DTF holds for the children of this node.

```
boolean swapChildrenAsNeeded() {
    if (dtfLeft() < dtfRight()) {
        LeftistHeapNode<E> temp = left;
        left = right;
        right = temp;
        dtf = 1 + dtfRight();
        return true;
    }
    return false;
}
```

Correctness Highlights: By the requirement, DTF holds with respect to *left* and *right*. Thus the conditional preserves LEFTLEANING. By LEFTLEANING and the definition of *dtf*, the last line preserves DTF for this node. As noted, this method may violate these properties for the parent node, and it is the caller's obligation to restore them. ■

26.4 Leftist Heap Methods

In this section we present internal and public methods for the LeftistHeap class.

26.4.1 Constructors

The constructor that takes no parameters creates an empty leftist heap that uses the default comparator.

```
public LeftistHeap(){
    this(Objects.DEFAULT_COMPARATOR);
}
```

The constructor that takes *comp*, a user-provided comparator, creates an empty leftist heap that uses the given comparator to define an ordering among the elements. When a new leftist heap is created, *proxy* is initialized to a union-find node referencing itself.

```
public LeftistHeap(Comparator<? super E> comp) {
    super(comp);
    proxy = new UnionFindNode<LeftistHeap<E>>(this);
}
```

The constructor that takes *h1*, a leftist heap and *h2*, a leftist heap, creates a new leftist heap which holds all elements from *h1* and all elements from *h2*. That is, a new leftist heap is created that holds the union of *h1* and *h2*. It throws an *IllegalArgumentException* when *h1* and *h2* do not use the same comparator. All locators for elements in *h1* and *h2* remain valid for their new position in the newly created heap. As a side effect *h1* and *h2* are empty after this method completes. However, new elements can be added to *h1* and *h2* to repopulate either of them.

```
public LeftistHeap(LeftistHeap<E> h1, LeftistHeap<E> h2) {
    this(h1.comp);                                //create a new leftist heap with comparator from h1
    if (h2.comp != h1.comp)                        //check h1 and h2 use the same comparator
        throw new IllegalArgumentException
            ("The comparators of the two heaps must be the same.");
    root = merge(h1.root, h2.root);                //root set to merge of h1 and h2
    size = h1.size + h2.size;                      //size of new heap is sum of sizes of h1 and h2

    //The remainder of this method ensures that locators remain valid through our use of
    //the union-find data structure
    proxy.union(h1.proxy);                        //take union of proxies of new heap and h1
    proxy.union(h2.proxy).set(this);              //rep. element for union of this and h2 is this
}
```

```

        h1.reinitialize();
        h2.reinitialize();
    }

```

*//reinitialize h1 and h2
//so new elements can be added to them*

Correctness Highlights: By the correctness of *merge*, the leftist heap referenced by *root* satisfies HEAPORDERED, LEFTLEANING, DTF, REACHABLE, PARENT, and INUSE. Since *SIZE* holds for *h1* and *h2*, setting *size* to the sum of *h1.size* and *h2.size* preserves *SIZE* for the newly created leftist heap. By the correctness of *reinitialize*, all properties hold for *h1* and *h2* where both are now empty leftist heaps. By the correctness of the union-find data structure (as discussed in Section 6.3), the proxy for *h1*, *h2*, and the newly created heap all reference the representative element for the component that includes all three. ■

The *reinitialize* method reinitializes all parameters so this leftist heap is empty.

```

void reinitialize() {
    size = 0;
    root = null;
    proxy = new UnionFindNode<LeftistHeap<E>>(this);
}

```

Correctness Highlights: Setting *size* to 0 preserves *SIZE*. By setting *root* to *null*, *REACHABLE* is preserved. The remaining properties hold vacuously since there are no elements in the heap. ■

26.4.2 Algorithmic Accessors

The internal *contains* method takes *target*, the target value, and *x*, the root of the subtree to search. It returns *true* if and only if $T(x)$ contains an element equivalent to the target.

```

protected boolean contains(E target, LeftistHeapNode<E> x){
    return x != null &&
        (equivalent(x.element, target) ||
         contains(target, x.left) ||
         contains(target, x.right));
}

```

The public *contains* method takes *target*, the target value. It returns *true* when there is an equivalent element to the target in this collection. Otherwise, *false* is returned.

```

public boolean contains(E target) {
    return contains(target, root);
}

```

Correctness Highlights: Follows from *REACHABLE* and the correctness of the internal *contains* method. ■

The *max* method returns a highest priority element. It throws a *NoSuchElementException* when this leftist heap is empty.

```

public E max() {
    if (isEmpty())
        throw new NoSuchElementException();
    return root.element;
}

```

Correctness Highlights: Follows directly from `HEAPORDERED` since all descendants of the root have an equal or lower priority. ■

A **preorder traversal** is a tree traversal that visits the current node, then recursively traverses the left subtree, and finally recursively traverses the right subtree. The *traverseForVisitor* method that takes *v*, the visitor, and *x*, a reference to root of the subtree to visit uses a preorder traversal to visit $T(x)$. We choose a preorder traversal since it visits the element at the root first. Any exception thrown by the visitor propagates to the calling method.

```

void traverseForVisitor(Visitor<? super E> v, LeftistHeapNode<E> x) throws Exception {
    if (x != FRONTIER) {
        v.visit(x.element);
        traverseForVisitor(v, x.left);
        traverseForVisitor(v, x.right);
    }
}

```

Correctness Highlights: All elements in $T(x)$ are visited exactly once. ■

The method *traverseForVisitor* takes *v*, the visitor, and applies the visitor from the root of the leftist heap. By `HEAPORDERED`, a highest priority element is always visited first.

```

protected void traverseForVisitor(Visitor<? super E> v) throws Exception {
    traverseForVisitor(v, root);
}

```

Correctness Highlights: By `REACHABLE` and the correctness of the *traverseForVisitor* method, all vertices are visited once each. ■

The *writeElements* method supports the implementation of *toString* by taking *sb*, a string buffer in which to create a string representation of the collection. The method appends each item's string representation to the string buffer, separated by commas. To perform the traversal more efficiently, as with *traverseForVisitor*, the implementation uses a visitor, which visits the elements of the collection using an preorder traversal.

```

protected void writeElements(final StringBuilder sb) {
    if (!isEmpty()) { //only visit the collection if it is non-empty
        accept(new Visitor<E>() {
            public void visit(E item) throws Exception {
                sb.append(item);
                sb.append(", ");
            }
        })
    }
}

```

```

    });
    int extraComma = sb.lastIndexOf(", ");    //remove the comma and space characters
    sb.delete(extraComma, extraComma+2);      //after the last element
}
}
}

```

Although there is no efficient search method for a leftist heap, we include a linear time internal method to locate an element. As with *traverseForVisitor*, a preorder traversal is used. The internal recursive *find* method that takes *x*, a reachable heap node, and *element*, the target. It returns a reference to a heap node holding an equivalent element, or *null* if there is no equivalent element in $T(x)$. This method performs the optimization of searching in $T(x)$ only if the target has a lower priority than the element in *x*.

```

LeftistHeapNode<E> find(LeftistHeapNode<E> x, E element){
    if (x == FRONTIER || comp.compare(element, x.element) > 0)
        return null;
    if (equivalent(element, x.element))                //check if x holds an equivalent element
        return x;
    LeftistHeapNode<E> fromLeft = find(x.left, element);    //recursively search left subtree
    if (fromLeft != null)                                //if found in left subtree
        return fromLeft;                                //return location found
    else                                                //otherwise
        return find(x.right, element);                  //return location found in right subtree
}

```

Correctness Highlights: By REACHABLE, all elements in $T(x)$ are compared with *element* unless it is determined, by HEAPORDERED that they are all too small. If a heap node in $T(x)$ holds an element equivalent to *x*, then a reference to that heap node is returned. A return value of *null* can occur only when no node in $T(x)$ holds an element equivalent to *element*. ■

The *find* method that takes *element*, the target, returns a reference to a heap node holding an equivalent element, or *null* if there is no equivalent element in this collection. This method starts the recursive *find* method at the root.

```

protected LeftistHeapNode<E> find(E element){
    return find(root, element);
}

```

Correctness Highlights: By the correctness of the recursive *find* method, returns *null* only when there is no equivalent element in $T(\text{root})$. Otherwise, it returns a reference to a reachable element equivalent to *element*. ■

The method *getEquivalentElement* takes *target*, the target element, and returns an equivalent element. It throws a *NoSuchElementException* when there is no equivalent element in this collection.

```

public E getEquivalentElement(E target) {
    LeftistHeapNode<E> node = find(target);
    if (node != FRONTIER)
        return node.element;
    else

```

```

        throw new NoSuchElementException();
    }

```

26.4.3 Content Mutators

The workhorse for all content mutators is the internal *merge* method. When either one of the two heaps to merge is empty, the process is straightforward. Otherwise, the node among *h1* and *h2* having a higher priority (with ties arbitrarily broken in favor of *h1*), becomes the root. To simplify the code, we always set *h1* to be the heap node among *h1* and *h2* which becomes the root, and set *h2* to be the other. The merge process proceeds by replacing the right subtree of *h1* by the leftist heap that results when recursively merging the current right subtree of *h1* and *h2*. Then *swapChildrenAsNeeded* is used to restore DTF and LEFTLEANING. Finally, all mutations that would interfere with the ability for the visiting iterator to continue properly, call *merge*. Thus by invalidating all active locators in this method, the modification count is updated when needed.

The *merge* method that takes *h1*, a reference to the root of one leftist heap and *h2*, a reference to the root another leftist heap. It combines them into a single leftist heap, and returns a reference to the root of the result leftist heap. This method requires that *T(h1)* and *T(h2)* are valid leftist heaps. More specifically *T(h1)* and *T(h2)* must satisfy HEAPORDERED, LEFTLEANING, DTF, PARENT, and INUSE. This method guarantees that the resulting heap includes each element from *T(h1)* and *T(h2)* exactly once. It also preserves HEAPORDERED, LEFTLEANING, DTF, PARENT, and INUSE.

```

LeftistHeapNode<E> merge(LeftistHeapNode<E> h1, LeftistHeapNode<E> h2) {
    if (h2 == FRONTIER) return h1;           //if one root is null,
    if (h1 == FRONTIER) return h2;           //then return the other
    if (comp.compare(h1.element, h2.element) < 0) { //if h2 has higher priority
        LeftistHeapNode<E> temp = h1;        //swap their roles
        h1 = h2;
        h2 = temp;
    }
    h1.setRight(merge(h1.right, h2));         //recursively merge h1.right and h2
    version.increment();                       //invalidate all locators
    return h1;
}

```

Correctness Highlights: Both *T(h1)* and *T(h2)* are valid leftist heaps and thus preserve all leftist heap properties. When *T(h1)* or *T(h2)* are empty, clearly all properties are preserved, and the returned value is correct.

Since the root of *h1* has priority at least as large as that of the root of *h2*, and both *h1* and *h2* obey HEAPORDERED, HEAPORDERED is preserved in the returned heap. PARENT is preserved by *setRight*, which calls *swapChildrenAsNeeded*. By the correctness of *swapChildrenAsNeeded*, DTF and LEFTLEANING are preserved. Since no left child is modified, INUSE is preserved. Finally, using an inductive argument, by the correctness of *merge* (for a smaller size input), the right subtree of *h1* preserves all of the leftist heap properties. Thus it also follows that all elements in *T(h1)* and in *T(h2)* appear exactly once in the merged leftist heap since every element in the left subtree of *h1* remains in the left subtree, and every element in the right subtree of *h1* and in all of *h2* will be in the right subtree of *h1*. ■

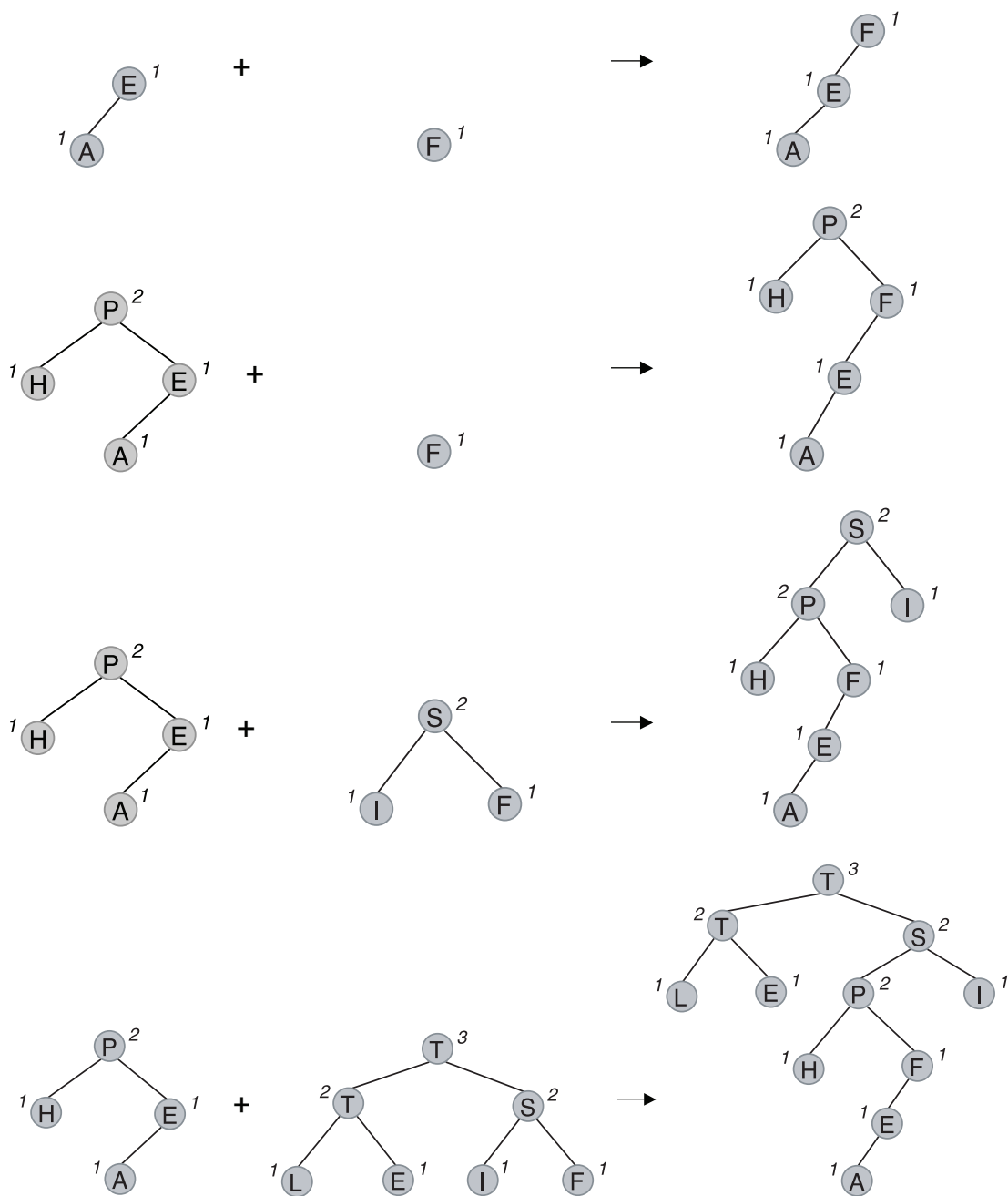


Figure 26.2

Illustration of a method to merge two leftist heaps. Recall that the number shown by each heap node is its shortest distance to a frontier. Each row shows the result of merging two leftist heaps. Furthermore, each row except for the first, recursively performs the merge shown in the row above it.

Methods to Perform Insertion

To insert a new element into a leftist heap, a singleton leftist heap x is created that just holds the new element. Then x is combined with the leftist heap holding the other elements in the collection using *merge*. Observe that when the new element has a higher priority than all existing elements, it becomes the new root, and the existing leftist heap becomes its left child. A sequence of insertions is shown in [Figure 26.3](#).

The internal *insert* method takes *element*, the new element to insert into this collection. It returns a reference to the heap node holding the new element.

```
LeftistHeapNode<E> insert(E element) {
    LeftistHeapNode<E> x = new LeftistHeapNode<E>(element);
    root = merge(root, x);
    size++;
    return x;
}
```

//preserve Size

Correctness Highlights: By the correctness of *merge*, HEAPOORDERED, DTF, LEFTLEANING, PARENT, INUSE, and REACHABLE are preserved. Incrementing *size* preserves SIZE. ■

The *add* method takes *element*, the element to add.

```
public void add(E element) {
    insert(element);
}
```

Correctness Highlights: Follows from the correctness of *insert*. ■

The *addTracked* method *element*, the element to add. It returns a tracker to the newly added element.

```
public PriorityQueueLocator<E> addTracked(E element) {
    return new Tracker(insert(element), proxy);
}
```

Correctness Highlights: Follows from the correctness of *insert* and the Tracker constructor. ■

Methods to Modify Elements

As with insertion, all other mutators rely heavily on *merge*. The basic approach used to either update the value in a node, or to remove a node, starts by detaching a portion of the leftist heap to form a “new” leftist heap. For example, [Figure 26.4](#) shows the leftist heap that results when detaching the subtree root at r from the leftist heap shown at the top of the figure. Then, the portions of the leftist heap that hold elements that should still be in the collection, are reintroduced into $T(\text{root})$ using *merge*.

The *detachFromParent* method takes x , a reference to a heap node for which $T(x)$ is to be detached. It breaks the leftist heap into two heaps by making $T(x)$ its own leftist heap. This method requires that the given node is not the root. All properties, except SIZE, are preserved by this method.

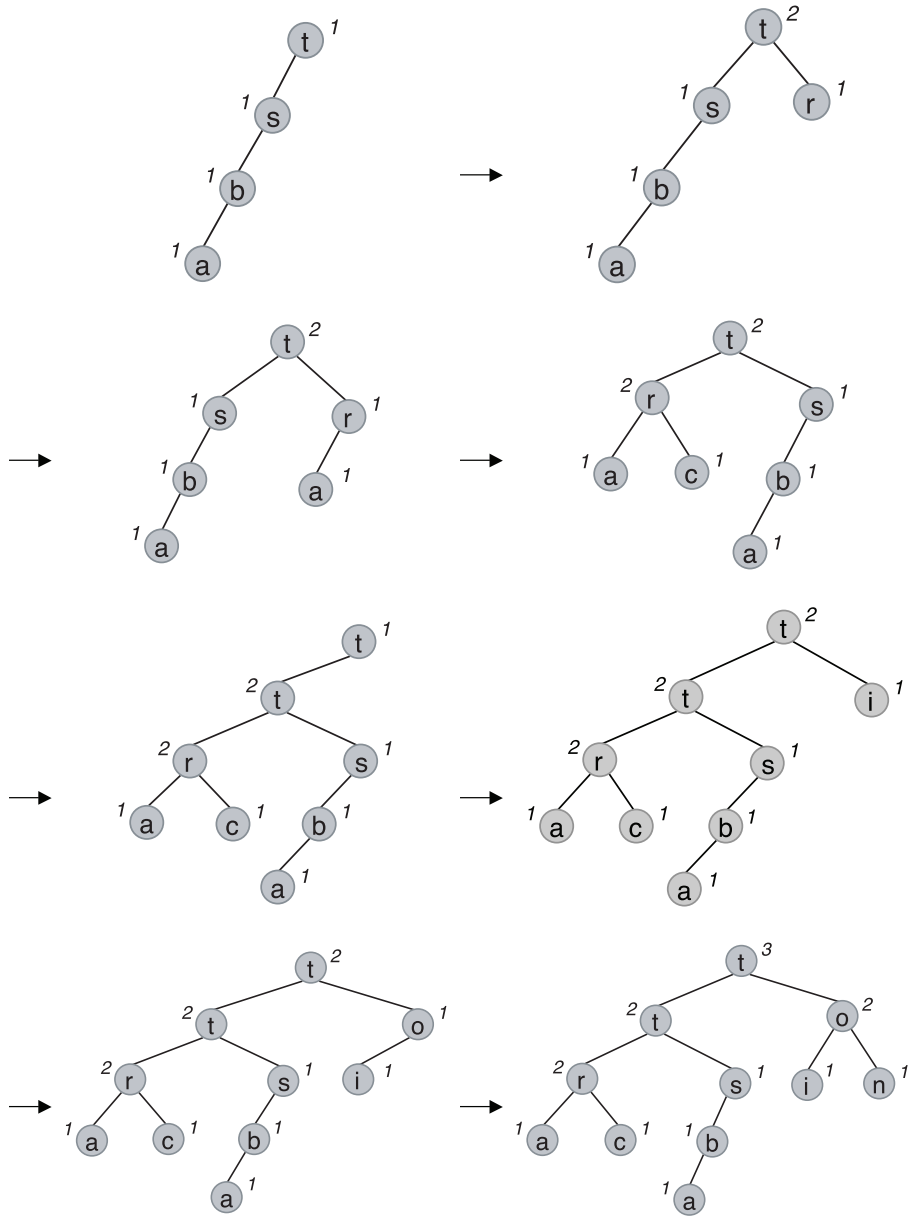


Figure 26.3

Illustration of adding the letters in “abstraction,” in that order, into an initially empty leftist heap. Whenever the new element is larger than all other elements in the collection, it becomes the new root with the current leftist heap as its left child and a *null* right child. The first leftist heap shown is the one obtained after inserting “abst.”

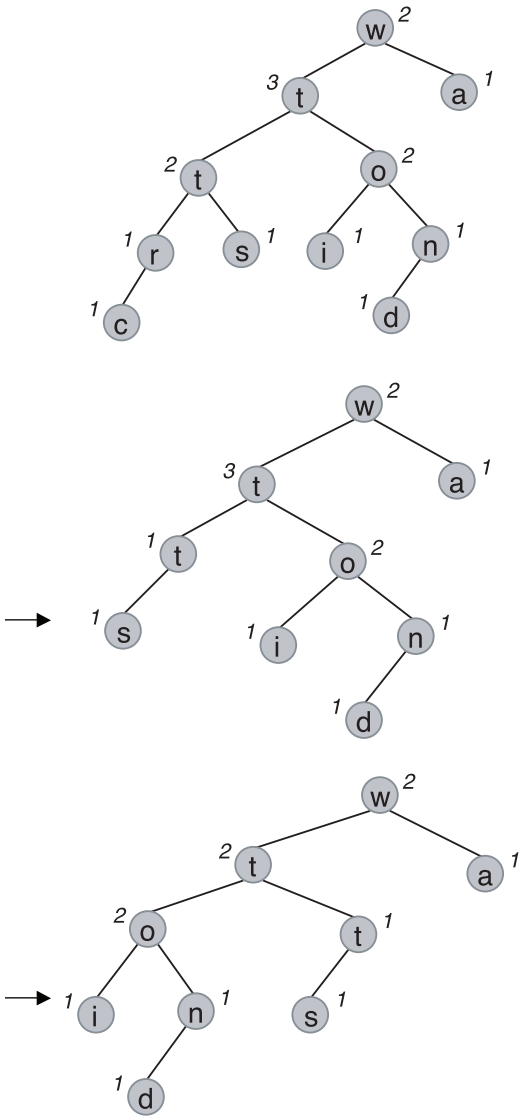


Figure 26.4
The intermediate steps when detaching, and removing the subtree rooted at *r* from the leftist heap at the top of the figure. The middle row shows how the children of the subtree rooted at the *t* with a dtf value of 2 are interchanged to preserve LEFTLEANING at that node. Likewise, the last row shows the changes made to preserve LEFTLEANING at *t* with a dtf value of 3.

```

void detachFromParent(LeftistHeapNode<E> x) {
    if (x.parent.left == x)                                     //if node is a left child
        x.parent.left = null;
    else                                                         //if node is a right child
        x.parent.right = null;
    LeftistHeapNode<E> ptr = x.parent;                          //preserve DTF and LeftLeaning
    x.parent = null;                                           //now node is detached from parent
    while (ptr != null && ptr.swapChildrenAsNeeded())
        ptr = ptr.parent;                                     //move up one level in heap
}

```

Correctness Highlights: First x is detached by both setting the appropriate child reference of x 's parent to x to **null**. Also, x is made the root of the detached leftist heap by setting its parent to **null**. Observe that PARENT, REACHABLE, and INUSE are preserved for all the heap nodes in $T(\text{root}) - T(x)$. However, when x was a left child, LEFTLEANING could be violated.

We now argue that the *while* loop restores DTF and LEFTLEANING for the nodes that remain in $T(\text{root})$. Since these properties held before $T(\text{node})$ was detached, the only possible violation would be with respect to ptr which is initially the parent of node . Furthermore, either the instance variable dtf is correct for the children of ptr or the child is **null**. By the correctness of *swapChildrenAsNeeded*, DTF and LEFTLEANING are restored with respect to ptr . Since dtf could change for ptr , it is possible that LEFTLEANING is violated for the parent of ptr , and so the *while* loop continues at ptr.parent until either the *root* is processed or a node is reached along the path to the root for which LEFTLEANING holds. No other properties, except SIZE, are affected for $T(\text{root})$. ■

The method *decreasePriority* takes *node*, a reference to the heap node holding element e and *element*, the new element to replace e . This method requires that $\text{element} \leq \text{node.element}$. The approach taken in this method is to detach any child that has a larger priority than the updated priority for *element*. Then *merge* is used to combine the detached heap(s) with $T(\text{root})$.

```

void decreasePriority(LeftistHeapNode<E> node, E element) {
    node.element = element;                                     //replace element in node
    LeftistHeapNode<E> leftChild = node.left;
    LeftistHeapNode<E> rightChild = node.right;
    if (leftChild != FRONTIER &&                               //if left child exists and has higher priority
        comp.compare(leftChild.element, element) > 0)
        detachFromParent(leftChild);                          //detach it from current location
    else
        leftChild = null;                                       //otherwise, remember no left child to merge back
    if (rightChild != FRONTIER &&                               //if right child exists and has higher priority
        comp.compare(rightChild.element, element) > 0)
        detachFromParent(rightChild);                          //detach it from current location
    else
        rightChild = null;                                       //otherwise, remember no right child to merge back
    root = merge(root, leftChild);                             //merge with T(root)
    root = merge(root, rightChild);                             //merge with T(root)
}

```

Correctness Highlights: Since any child of *node* that had a element with a higher priority than *element* was detached and then merged back into $T(\text{root})$, HEAPORDERED is preserved. Observe

that the number of elements in $T(\text{root})$ at the end is exactly the number at the start, so SIZE is preserved. The rest of the correctness follows from that of *detachFromParent* and *merge*. ■

The method *increasePriority* takes *node*, a reference to the heap node holding element *e* and *element*, the new element to replace *e*. This method requires that $\text{element} \geq \text{node.element}$ and that *node* is not *null*. The approach taken here is to detach *node* from its parent if the update would cause a violation of HEAPOORDERED. Then *merge* is used to recombine the detached heap with $T(\text{root})$.

```
void increasePriority(LeftistHeapNode<E> node, E element){
    node.element = element;                                //replace element
    if (node != root &&                                     //if node is not the root and
        comp.compare(node.parent.element, element) < 0){   //HeapOrdered violated
        detachFromParent(node);
        root = merge(root, node);
    }
}
```

Correctness Highlights: If *node* is the root then all that is needed is to update the value. No properties could be violated here. Otherwise, if the element in *node*'s parent has a lower priority than *element*, then $T(\text{node})$ is detached and then merged back into $T(\text{root})$, preserving HEAPOORDERED. The number of elements in $T(\text{root})$ at the end is exactly the number there at the start, so SIZE is preserved. The rest of the correctness follows from that of *detachFromParent* and *merge*. ■

Finally, the internal *update* method replaces the element held by a node. It takes *node*, a reference to node where element is to be replaced, and *element*, the replacement element. This method requires that *node* is not *null*. Figure 26.5 illustrates a sequence of updates.

```
void update(LeftistHeapNode<E> node, E element){
    if (comp.compare(element, node.element) ≥ 0)
        increasePriority(node, element);
    else if (comp.compare(element, node.element) < 0)
        decreasePriority(node, element);
}
```

Methods to Perform Deletion

In general to remove node *x* from a leftist heap, node *x* is detached from its parent, and then both of its children are merged back into the $T(\text{root})$. When *node* has no children, the locators are not invalidated. By LEFTLEANING, when *x* has a single child, it must be a left child. It might seem simplest to just replace *x* by its child. However, this modification could cause a violation of DTF and LEFTLEANING since *dtf* of node's parent could be changed by the removal of *node*. While a process like that of *merge* could be used to restore these two properties, we instead directly use *merge* when removing has a single child.

The internal *remove* method takes *x*, a reference to the heap node to remove, and removes *x* from the leftist heap. It returns the element held in *x*. This method requires that *x* is not *null*.

```
E remove(LeftistHeapNode<E> x) {
    if (x == root)
        root = merge(root.left, root.right);
    else {
        detachFromParent(x);                                //detach node from parent
    }
}
```

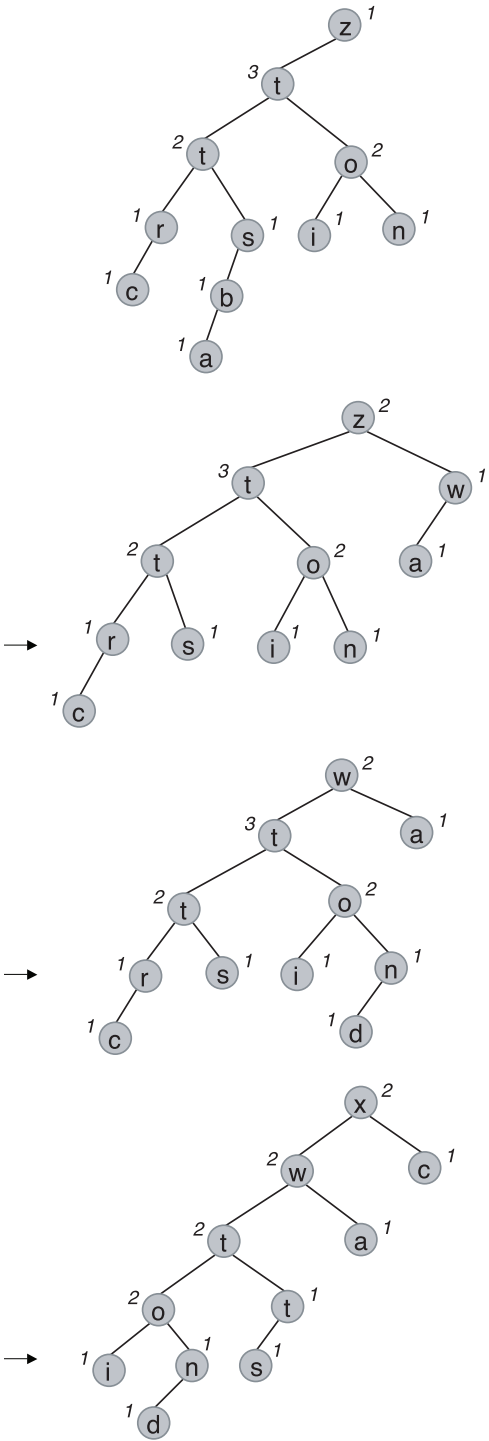


Figure 26.5

An illustration of three calls to *update* (through a tracker) for the leftist heap shown in Figure 26.1. First one of the “a”s is replaced by “z.” Observe that if the locator position is set with *getLocator*(“a”), then this will be the element updated since it occurs first in an inorder traversal of the leftist heap. Next “b” is replaced by “w.” Then the “z” is replaced by “d,” and finally one of the “t”s is replaced by an “x.”

```

        root = merge(root, x.left);           //merge left child into T(root)
        root = merge(root, x.right);        //merge right child into T(root)
    }
    size--;                                //preserve Size
    x.markDeleted();                       //preserve InUse
    return x.element;
}

```

Correctness Highlights: When x is the root, its children are merged to form the new root. Otherwise, x is separated from $T(\text{root})$. Thus by the correctness of *detachFromParent* and *merge*, **HEAPORDERED**, **DTF**, **LEFTLEANING**, **REACHABLE**, and **PARENT** are preserved. By the correctness of *markDeleted*, **INUSE** is preserved. Finally, decrementing *size* preserves **SIZE**. ■

The public *remove* method takes *target*, the element to remove. It removes an equivalent element from the collection. It returns *true* if and only if an equivalent element was found and removed.

```

public boolean remove(E target) {
    LeftistHeapNode<E> node = find(target);
    if (node == FRONTIER)
        return false;
    remove(node);
    return true;
}

```

The *extractMax* method returns the highest priority element from the collection. This method also removes the maximum priority element from the collection. It throws a *NoSuchElementException* when the collection is empty.

```

public E extractMax() {
    if (isEmpty())
        throw new NoSuchElementException();
    return remove(root);
}

```

The internal recursive *clear* method takes x , a reachable heap node, and removes all elements in $T(x)$. It uses a **postorder traversal** that first recursively clears the left subtree, then recursively clears the right subtree, and then removes x .

```

void clear(LeftistHeapNode<E> x) {
    if (x != FRONTIER) {
        clear(x.left);
        clear(x.right);
        x.markDeleted();
    }
}

```

Correctness Highlights: Follows by the correctness of *markDeleted*, and the observation that for any heap node $y \in T(x)$, once the left and right subtrees have had all elements removed, *clear*(y) is executed. ■

The public *clear* method removes all elements from the collection. This method starts a recursive clear method at the root.

```

public void clear() {
    clear(root);
    size = 0;
    root = FRONTIER;
}

```

Correctness Highlights: By the correctness of the recursive *clear* method, all elements in $T(\text{root})$ are removed from this leftist heap. Finally, *size* is reset to preserve SIZE, and *root* is reset to preserve REACHABLE. ■

26.4.4 Locator Initializers

The *iterator* method creates a new tracker that is at FORE.

```

public Locator<E> iterator() {
    return new VisitingIterator();
}

```

The method *getLocator* takes *element*, the target. It returns a tracker positioned at the given element. It throws a *NoSuchElementException* when there is no equivalent element in this collection.

```

public PriorityQueueLocator<E> getLocator(E element) {
    LeftistHeapNode<E> node = find(element);
    if (node == null)
        throw new NoSuchElementException();
    else
        return new Tracker(node, proxy);
}

```

Correctness Highlights: Correctness follows from *find* and the Tracker constructor, which takes a reference to the heap node to track, and the proxy for the leftist heap on which this method was called. ■

26.5 Tracker Inner Class

AbstractCollection<E>.AbstractLocator<E> implements *Locator*<E>

└ **Tracker** implements *PriorityQueueLocator*<E>

Each tracker references the heap node it tracks, and also includes a union-find node proxy for the leftist heap to which this tracker belongs. Including the proxy allows trackers to remain valid after two heaps are merged as discussed in much more depth in Section 6.8.

```

LeftistHeapNode<E> node;

```

```
UnionFindNode<LeftistHeap<E>> proxy;           //preserves trackers after merge
```

The constructor takes *node*, a reference to the node to track, and *proxy*, the union-find node that serves as a proxy for the leftist heap to which this tracker belongs.

```
Tracker(LeftistHeapNode<E> node, UnionFindNode<LeftistHeap<E>> proxy) {
    this.node = node;
    this.proxy = proxy;
}
```

Recall that *inCollection* returns *true* if and only if the tracked element is currently in the collection.

```
public boolean inCollection() {
    return !node.isDeleted();
}
```

The *get* method returns the tracked element. It throws a *NoSuchElementException* when the tracker is at FORE, at AFT, or is tracking an element that has been removed.

```
public E get() {
    if (node.isDeleted())
        throw new NoSuchElementException();
    return node.element;
}
```

Since a visiting iterator (versus a threaded implementation) is provided as the iterator implementation, *advance*, *retreat*, and *hasNext* are unsupported operations for the tracker (which is used only to track, update, and remove elements).

```
public boolean advance() {
    throw new UnsupportedOperationException();
}

public boolean retreat() {
    throw new UnsupportedOperationException();
}

public boolean hasNext() {
    throw new UnsupportedOperationException();
}
```

The *remove* method removes the tracked element from the collection.

```
public void remove() {
    if (node.isDeleted())
        throw new NoSuchElementException();
    proxy.findRepresentative().get().remove(node);
}
```

Correctness Highlights: The union-find *findRepresentative* method returns the representative union-find node for the leftist heap that currently holds *node*. Also by the correctness of the union-find *get* method, it returns a reference to the leftist heap to which *node* currently belongs. The rest of the correctness follows from that of the heap node *isDeleted* and *remove* methods. ■

The *update* method replaces the tracked element by *element*.

```
public void update(E element) {
    if (node.isDeleted())
        throw new NoSuchElementException();
    proxy.findRepresentative().get().update(node, element);
}
```

Correctness Highlights: Analogous to the correctness discussion for *remove*. ■

The *increasePriority* method replaces the tracked element by *element*. It requires that the given parameter is greater than *e*, or that *e* is the parameter being passed and its value has been mutated to have a higher priority than it had previously. That is, it is acceptable practice to mutate the element to have a higher priority and then immediately call *increasePriority* to restore the properties of the priority queue.

```
public void increasePriority(E element) {
    if (node.isDeleted())
        throw new NoSuchElementException();
    proxy.findRepresentative().get().increasePriority(node, element);
}
```

Correctness Highlights: Analogous to the correctness discussion for *remove*. ■

The *decreasePriority* method replaces the tracked element by *element*. It requires that the given parameter is less than *e*, or that *e* is the parameter being passed and its value has been mutated to have a lower priority than it had previously. That is, it is acceptable practice to mutate the element to have a lower priority and then immediately call *decreasePriority* to restore the properties of the priority queue.

```
public void decreasePriority(E element) {
    if (node.isDeleted())
        throw new NoSuchElementException();
    proxy.findRepresentative().get().decreasePriority(node, element);
}
```

Correctness Highlights: Analogous to the correctness discussion for *remove*. ■

26.6 Performance Analysis

The asymptotic time complexities of all public methods for the LeftistHeap class are shown in Table 26.6, and the asymptotic time complexities for all of the public methods of the LeftistHeap Tracker class are given in Table 26.7.

Observe that the space complexity requires allocating one heap node for every element. For an unthreaded leftist heap there are five instance variables (*element*, *left*, *right*, *parent*, and *dtf*). There are two additional instance variables added for a threaded version (*next* and *prev*).

method	time complexity
<i>ensureCapacity(x)</i> <i>max()</i>	$O(1)$ $O(1)$
<i>add(o)</i> <i>addTracked(o)</i> <i>extractMax()</i>	$O(\log n)$ $O(\log n)$ $O(\log n)$
<i>clear()</i> <i>contains(o)</i> <i>getEquivalentElement(o)</i> <i>getLocator(o)</i> <i>remove(o)</i> <i>toArray()</i> <i>toString()</i> <i>trimToSize()</i> <i>accept()</i>	$O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$
<i>addAll(c)</i>	$O(c \log n)$
<i>retainAll(c)</i>	$O(c n)$

Table 26.6 Summary of the asymptotic time complexities for the *PriorityQueue* public methods when implemented using a leftist heap.

locator method	complexity
<i>get()</i> <i>next()</i>	$O(1)$ $O(1)$, amortized
<i>update(o)</i> , increase priority <i>update(o)</i> , decrease priority <i>remove()</i>	$O(\log n)$ $O(\log n)$ $O(\log n)$
<i>advance()</i> <i>hasNext()</i> <i>retreat()</i>	unsupported unsupported unsupported

Table 26.7 Summary of the time complexities for the locator methods of the leftist heap tracker.

The space requirements to support the ability to merge two leftist heaps is very small. By making the `HeapNode` class static, *this* is not needed, but instead the reference *proxy* to a union-find node is used. Thus, there is no change in the space for each heap node. The only additional cost is the one union-find node that is allocated each time the constructor is called, and three union-find nodes are allocated each the constructor is called that takes two leftist heaps. There are the union-find node allocated for the new leftist heap, and a new union-find node is allocated for each of the old heaps in order to allow them to be repopulated.

Since the tracker constructor takes constant time, so does *iterator*. Also, since the leftist heap is an elastic implementation, *ensureCapacity* and *trimToSize* need not perform any computation, so they take constant time.

The highest priority element is always in the root, so it can be found in constant time. Thus *max* takes constant time. For all other elements, while *find* performs an optimization to minimize the costs, in the worst-case the target element must be compared to all elements in the collection. Thus, in the worst-case, *find*, *getEquivalentElement*, and *getLocator* take linear time.

We now explore the relationship between n and $dtf(root)$. Let $n_x = |T(x)|$. That is n_x is the number of nodes in the subtree $T(x)$. The following two properties hold for any reachable node x in $T(root)$.

- $n_x \geq 2^{dtf(x)} - 1$,
- $dtf(root) \leq \log_2(n + 1)$, and
- the length of the path in $T(x)$ obtained by following the right child until reaching a frontier node has length $dtf(x)$.

To prove the first claim, observe that by the definition of $dtf(x)$, the first $dtf(x)$ levels of the heap are full. Thus $n_x \geq 1 + 2 + 4 + \dots + 2^{dtf(x)-1} = 2^{dtf(x)} - 1$. The second claim follows from the first claim by letting $x = root$, and solving for $dtf(root)$. Finally, the third claim follows from the fact that for a frontier node, dtf is 0, and by the definition of dtf and the property DTF, $dtf(x) = dtf(x.right) + 1$.

Since the *merge* method has time proportional to the path obtained when following the right child, it follows that the time complexity is proportional to $dtf(root) = O(\log n)$. Thus *merge*, *add*, and *addTracked* have logarithmic time complexity. Likewise, once a node to remove or update has been located, each execution of *remove* and *update* require just one or two calls of *merge* so they also have time complexity $O(\log n)$.

The *clear*, *toString*, *toArray*, and *accept* methods perform a recursive tree traversal that visits each node exactly once, spending constant time at each. So the overall cost is $O(n)$.

When the parameter for *addAll* is a leftist heap, then the logarithmic time *merge* method is used. Otherwise, each of the elements in collection c is inserted, which takes logarithmic time per element, leading to an overall cost of $O(|c| \log n)$.

The *retainAll* method performs a search for each element in the provided collection c . Any element not found in c is removed. In the worst-case, for each element a linear time search must be performed, which is the dominant cost since removing an element (once located) takes $O(\log n)$ time.

We now analyze the time complexity of the locator methods. The *get*, *remove*, and *update* methods take constant time to locate the element. For *get*, the element is returned in constant time. For *update* and *remove*, the mutation is performed by the corresponding internal methods, both of which take logarithmic time. Finally, *next* is performed through the visiting iterator which takes $O(n)$ time to iterate through the entire collection since it uses a preorder traversal. Thus the amortized cost for each call to *next* is constant.

26.7 Quick Method Reference

LeftistHeap Public Methods

- p. 378** *LeftistHeap*()
- p. 378** *LeftistHeap*(Comparator<? super E> comp)
- p. 378** *LeftistHeap*(LeftistHeap<E> h1, LeftistHeap<E> h2)
- p. 98** void **accept**(Visitor<? super E> v)
- p. 384** void **add**(E element)
- p. 100** void **addAll**(Collection<? extends E> c)
- p. 384** PriorityQueueLocator<E> **addTracked**(E element)
- p. 100** void **clear**()
- p. 97** boolean **contains**(E value)
- p. 99** void **ensureCapacity**(int capacity)
- p. 390** E **extractMax**()
- p. 96** int **getCapacity**()
- p. 96** Comparator<? super E> **getComparator**()
- p. 98** E **getEquivalentElement**(E target)
- p. 391** PriorityQueueLocator<E> **getLocator**(E element)
- p. 96** int **getSize**()
- p. 96** boolean **isEmpty**()
- p. 391** Locator<E> **iterator**()
- p. 379** E **max**()
- p. 390** boolean **remove**(E target)
- p. 100** void **retainAll**(Collection<E> c)
- p. 97** Object[] **toArray**()
- p. 97** E[] **toArray**(E[] array)
- p. 98** String **toString**()
- p. 99** void **trimToSize**()

LeftistHeap Internal Methods

- p. 390** void **clear**(LeftistHeapNode<E> x)
- p. 97** int **compare**(E e1, E e2)
- p. 379** boolean **contains**(E target, LeftistHeapNode<E> x)
- p. 387** void **decreasePriority**(LeftistHeapNode<E> node, E element)
- p. 384** void **detachFromParent**(LeftistHeapNode<E> x)
- p. 97** boolean **equivalent**(E e1, E e2)
- p. 381** LeftistHeapNode<E> **find**(E element)
- p. 381** LeftistHeapNode<E> **find**(LeftistHeapNode<E> x, E element)
- p. 388** void **increasePriority**(LeftistHeapNode<E> node, E element)
- p. 384** LeftistHeapNode<E> **insert**(E element)
- p. 382** LeftistHeapNode<E> **merge**(LeftistHeapNode<E> h1, LeftistHeapNode<E> h2)
- p. 379** void **reinitialize**()
- p. 388** E **remove**(LeftistHeapNode<E> x)
- p. 99** void **traverseForVisitor**(Visitor<? super E> v)
- p. 380** void **traverseForVisitor**(Visitor<? super E> v, LeftistHeapNode<E> x)
- p. 388** void **update**(LeftistHeapNode<E> node, E element)
- p. 98** void **writeElements**(StringBuilder s)
- p. 380** void **writeElements**(StringBuilder sb)

LeftistHeap.LeftistHeapNode Public Methods

- p. 376** *LeftistHeapNode*(E element)

LeftistHeap.LeftistHeapNode Internal Methods

- p. 376** int **dtfLeft**()
- p. 377** int **dtfRight**()
- p. 376** boolean **isDeleted**()
- p. 376** void **markDeleted**()
- p. 377** void **setLeft**(LeftistHeapNode<E> x)
- p. 377** void **setRight**(LeftistHeapNode<E> x)
- p. 377** boolean **swapChildrenAsNeeded**()

LeftistHeap.Tracker Public Methods

- p. 392** boolean **advance**()
- p. 393** void **decreasePriority**(E element)
- p. 392** E **get**()
- p. 392** boolean **hasNext**()
- p. 101 void **ignoreConcurrentModifications**(boolean ignore)
- p. 101 void **ignorePriorConcurrentModifications**()
- p. 392** boolean **inCollection**()
- p. 393** void **increasePriority**(E element)
- p. 101 E **next**()
- p. 392** void **remove**()
- p. 392** boolean **retreat**()
- p. 393** void **update**(E element)

LeftistHeap.Tracker Internal Methods

- p. 392** *Tracker*(LeftistHeapNode<E> node, UnionFindNode<LeftistHeap<E>> proxy)
- p. 101 void **checkValidity**()
- p. 101 void **updateVersion**()

Chapter 27

Pairing Heap Data Structure

package `collection.priority`

`AbstractCollection<E>` implements *Collection<E>*

↳ `PairingHeap<E>` implements *PriorityQueue<E>*, *Tracked<E>*

Uses: Java references and Queue (Chapter 18)

Used By: `TaggedPairingHeap` (Section 49.8.4)

Strengths: Experimental studies [142, 104] have shown that, in practice, the pairing heap yields the best performance for many graph algorithms that rely upon a priority queue. (Chapter 56 discusses the greedy tree builder, from which these algorithms are derived.) In particular, Experiments by Moret and Shapiro [116] demonstrated that using a pairing heap within Prim's minimum spanning algorithm leads to a more efficient solution than using a binary heap, Fibonacci heap, or a variety of other priority queue data structures. Also, a pairing heap is a relatively simple, easily implemented, data structure. It is conjectured that the amortized cost of increasing the priority of an element is $O(\log \log n)$, although the best proven bound is $O(\log n)$.

Weaknesses: The worst-case cost when an element is removed or the priority is decreased is linear. Theoretical bounds are not as good as those for the Fibonacci heap. In particular, using a Fibonacci heap yields the theoretically best worst-case time complexities for Prim's minimum spanning tree algorithm and Dijkstra's shortest path algorithm.

Critical Mutators: Since our implementation is threaded, there are no critical mutators. In an unthreaded implementation *add*, *addAll*, *addTracked*, *extractMax*, *remove*, and *retainAll* would be critical mutators.

Competing Data Structures: The binary heap (Chapter 25) is much more space efficient, and generally a good competitor if there is no need to efficiently merge two priority queues, and if increasing the priority of elements is relatively rare. If there is a need to efficiently merge the data structures yet increasing the priority of elements is relatively rare, then a leftist heap (Chapter 26) is a good option. Finally, if it is important to have constant amortized cost for both merge and increasing the priority of an element, then the Fibonacci heap (Chapter 28) is the best option.

27.1 Internal Representation

The pairing heap is a heap-ordered tree, but not a binary tree. The tree is stored as a collection of heap nodes, where each heap node has references to one child, a left sibling, and a right sibling.

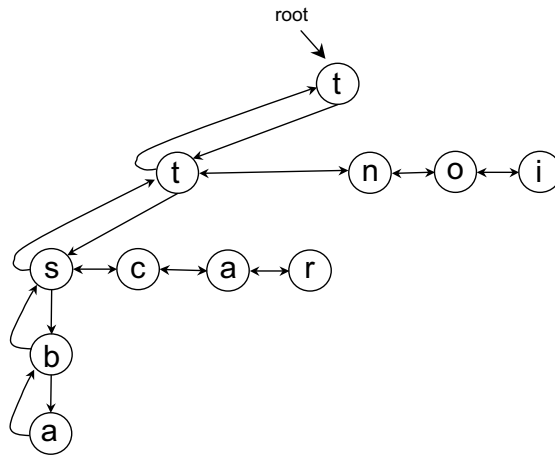


Figure 27.1

The internal representation for a populated example for a pairing heap holding the letters of “abstraction.” Horizontal arrows show the *sibL* and *sibR* sibling references, and downward arrows show the child references. The *next* and *previous* references that form the iteration list are not shown.

To reduce the space usage, there is no parent reference. Instead the child referenced by the parent, which we call the **leftmost sibling**, references its parent with its left sibling reference.

Instance Variables and Constants: In addition to the variables inherited from *AbstractCollection*, *root* refers to the root of the pairing heap. To provide efficient support for iteration, we use a **threaded implementation**, where each heap node has a *next* reference to the next element in the iteration order and a *prev* reference to the previous element in the iteration order. We refer to the list of elements defined by the next and previous references as the **iteration list**. We allocate a heap node *FORE* to serve as a sentinel head for the iteration list, and a heap node *AFT* to serve as the sentinel tail for the iteration list. The queue *detached*, as discussed in depth on page 410, is used as part of the process to update the priority of an element, or to remove an element. We include *detached* as an instance variable to avoid recreating a new queue each time it is needed.

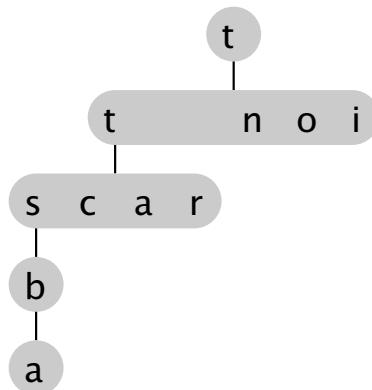


Figure 27.2

Our visual representation for the pairing heap shown in Figure 27.1.

```

HeapNode<E> root;
final HeapNode<E> FORE = new HeapNode<E>(null);
final HeapNode<E> AFT = new HeapNode<E>(null);
Queue<HeapNode<E>> detached = new Queue<HeapNode<E>>();

```

Populated Example: Figure 27.1 shows the internal representation for a pairing heap (excluding the iteration list) holding the letters of “abstraction.” It was obtained by inserting the letters in the word “abstraction” and then a “z” followed by a *extractMin*. Throughout this chapter we use a more compact visualization of a pairing heap, which shows all siblings together in a shaded rounded rectangle. The children of each element are connected by a line. Figure 27.2 shows the pairing heap from Figure 27.1 in this form. The order for the iteration list, which is based on the order in which the elements were added to the collection, is completely independent of the structure of the tree. The pairing heap visualizations do not show the iteration list.

Terminology:

- A **sibling chain** is a doubly linked list, using *sibL* and *sibR* references, containing all children of a common parent.
- The **leftmost sibling** in a sibling chain is the node x where $x.sibL.child = x$. Observe that for the leftmost sibling, $x.sibL$ references its parent. (These are shown as the up arrows in Figure 27.1.)
- We say that all nodes in a sibling chain have the **common parent** of $x.sibL$, where x is the leftmost sibling of the chain. For the root $x.sibL$ references itself. If $x.sibL = null$, then x has been removed from the collection.
- The **rightmost sibling** in a sibling chain is the unique node x , for which $x.sibR = null$.
- When a sibling chain with leftmost sibling y and rightmost sibling z is a child of node x , we say that y is the **leftmost child** of x , and that z is the **rightmost child** of x .
- Let x be a reference to a heap node. We define $succ(x)$ recursively as follows:

$$succ(x) = \begin{cases} AFT & \text{if } x = AFT \\ succ(x.next) & \text{if } x.sibL = null \\ x.next & \text{otherwise.} \end{cases}$$

We use this to define the location of each heap node in the iteration order. For any heap node x that holds an element in the collection, $succ(x)$ is the next item in the iteration list. For any heap node x that holds an element that has been removed, observe that the semantics of the tracker give that x is logically between $succ(x).prev$ and $succ(x)$.

- We let “ $\langle \rangle$ ” denote an empty sequence, and let “ $+$ ” denote the concatenation operator when applied to a sequence.
- For node x , we define

$$seq(x) = \begin{cases} \langle \rangle, & \text{if } x = AFT \\ seq(x.element) + seq(x.next) & \text{otherwise.} \end{cases}$$

Abstraction Function: Let PH be a pairing heap instance. The abstraction function is

$$AF(PH) = seq(FORE.next).$$

Optimizations: As with the leftist heap, there is a trade-off between providing efficient support for iteration and minimizing the space usage. Our threaded implementation of a pairing heap provides fast and robust support for iteration at the cost of introducing $2n$ additional references. As illustrated for the leftist heap (Chapter 26), one alternative approach is to use a visiting iterator to reduce the space complexity, at the cost of less efficient iteration.

The *decreasePriority* method could be optimized to detach only the children of the node being updated that cause a violation of `HEAPORDERED`.

27.2 Representation Properties

We inherit `SIZE` from `AbstractCollection`, and add the following properties. The first five properties are structural requirements on the pairing heap. The proper threading of the iteration order within the heap nodes is guaranteed by `ITERATIONLIST`. `REMOVED` identifies when a heap node holds an element that has been removed, and is used to argue that any tracker for a removed element has the specified behavior.

REACHABLE: An element is in the collection exactly when it is held within some heap node in $T(\text{root})$.

HEAPORDERED: The priority of each reachable heap node is at least as large as that of its descendants. That is the parent of a node has an element that is at least as large as its element. This implies that *root* references a maximum priority element.

CHILDREN: For each reachable heap node x , $x.child = \text{null}$ if and only if x has no children. Furthermore, when $x.child$ is not *null*, it references the leftmost child of x .

PARENT: For the root, $root.parent = root$, and for the leftmost sibling x of a sibling chain, $x.sibL.child = x$.

SIBLINGCHAIN: For each reachable heap node x , if $x.sibR \neq \text{null}$, $x.sibR.sibL = x$. Only one node x in the sibling chain, the rightmost sibling, has $x.sibR = \text{null}$.

ITERATIONLIST: Let x be any reachable node. Then $x.next$ references the next node in the iteration order (possibly *AFT*), and $x.prev$ references the previous node in the iteration order (possibly *FORE*).

REMOVED: A heap node $x \notin T(\text{root})$ if and only if $x.sibL = \text{null}$. In addition, any locator tracking x is logically between $\text{succ}(x).prev$ and $\text{succ}(x)$.

27.3 Heap Node Inner Class

HeapNode<*E*>

Each heap node includes a reference to its element, its leftmost child, its neighbors in the sibling chain, and the next and previous elements in the iteration list.

```

E element;
HeapNode<E> child;           //reference to the leftmost child
HeapNode<E> sibL;           //references left sibling in sibling list
HeapNode<E> sibR;           //references right sibling in sibling list
HeapNode<E> next;           //references next node in the iteration list
HeapNode<E> prev;           //references previous node in the iteration list

```

The constructor takes *element*, the element to be held in this node. Since *sibL* is *null*, the node is considered “deleted” until it is inserted into the heap.

```

public HeapNode(E element) {
    this.element = element;
    makeRoot();
}

```

The *makeRoot* method sets *sibL* and *sibR* so that this node becomes the root. Since the Java runtime system initializes *child* to *null* CHILDREN is satisfied.

```

void makeRoot() {
    sibL = this;           //satisfy Parent
    sibR = null;           //satisfy SiblingChain
}

```

Correctness Highlights: Setting *sibL* to itself satisfies PARENT, and setting *sibR* to *null*, satisfies SIBLINGCHAIN. ■

The *setLeft* method takes *ptr*, a reference to the heap node that is to be placed left of this node in the sibling chain.

```

final void setLeft(HeapNode<E> ptr) {
    sibL = ptr;
    ptr.sibR = this;
}

```

Correctness Highlights: This method helps to preserve SIBLINGCHAIN by ensuring that for all nodes *x* in the sibling chain except for the leftmost sibling, *x.sibL.sibR = x*. ■

The *setNext* method takes *ptr*, a reference to the node to place next in the iteration list.

```

final void setNext(HeapNode<E> ptr) {
    next = ptr;
    ptr.prev = this;
}

```

Correctness Highlights: This method preserves ITERATIONLIST. ■

The *isLeftmostChild* method returns *true* exactly when this node is a leftmost sibling.

```

final boolean isLeftmostChild() {
    return sibL.child == this;
}

```

Correctness Highlights: Follows from PARENT. ■

The *isRightmostChild* method returns *true* exactly when this node is a rightmost sibling.

```
final boolean isRightmostChild() {
    return sibR == null;
}
```

Correctness Highlights: Follows from SIBLINGCHAIN. ■

The *markDeleted* method modifies this node to indicate that the element it holds has been removed from the collection.

```
final void markDeleted() {
    sibL = null;
}
```

Correctness Highlights: This method preserves REMOVED with respect to the deleted node. ■

The *isDeleted* method returns *true* if and only if this occurrence of the element held is no longer in the collection.

```
final boolean isDeleted() {
    return sibL == null;
}
```

Correctness Highlights: Follows from REMOVED. ■

The *addChild* method takes *newChild*, a reference to the heap node to be added as a child to this node.

```
void addChild(HeapNode<E> newChild) {
    if (child == null) {                                //Case 1: has no children
        child = newChild;
        newChild.sibL = this;
    } else if (child.isRightmostChild()) {              //Case 2: has 1 child
        newChild.setLeft(child);
    } else {                                             //Case 3: has at least 2 children
        child.sibR.setLeft(newChild);                  //add as a second child in chain
        newChild.setLeft(child);
    }
}
```

Correctness Highlights: We argue that CHILDREN, PARENT, SIBLINGCHAIN, and REMOVED are preserved. We use *x* to denote the node on which this method is called.

Case 1: x currently has no children. By CHILDREN, this case occurs exactly when *child* is *null*. Setting *child* to *newChild* preserves CHILDREN, and setting *newChild.sibL* to *this* preserves both *Parent* and *InUse*. Finally, since *right* is initialized by the Java runtime system to *null*, SIBLINGCHAIN is preserved.

Case 2: x currently has one children. By CHILDREN and the correctness of *isRightmostChild*, when the leftmost and rightmost children are the same, x has a single child referenced by *child*. The two assignment statements combined with the fact that *newChild.sibR* is *null*, preserve SIBLINGCHAIN by making the existing child the leftmost child, and the new child the rightmost child. Also setting *newChild.sibL* to a value preserves REMOVED. By PARENT, *child.sibL.parent* = *this* before this method is called. No changes are made affecting this relationship, so PARENT.

Case 3: x currently has at least children. By CHILDREN, the new node is added just right of the leftmost child of this node. Since x has at least two children, by SIBLINGCHAIN, the leftmost child has a sibling to the right. The two calls to *setLeft* preserve SIBLINGCHAIN. Since the leftmost child is not changed, both CHILDREN and PARENT are preserved. ■

The *removeFromChain* method removes this heap node from its current sibling chain.

```
void removeFromChain() {
    if (isLeftmostChild())
        sibL.child = sibR;           //preserve Child
    else
        sibL.sibR = sibR;
    if (!isRightmostChild())        //preserve Parent
        sibR.sibL = sibL;
}
```

Correctness Highlights: Let x be the node on which this method is called, let p be its parent, and let r be its right sibling, or *null* if x is an only child. We consider the following cases.

Case 1: x is a leftmost child. By PARENT, *x.sibL* references p . Setting the child reference of p to r preserves CHILD. When x is an only child none of the other properties are affected. However, x has a right sibling, r becomes the leftmost child. So to preserve PARENT, *r.sibL* must be set to x 's left sibling (their parent).

Case 2: x is not a leftmost child. In this case CHILD is not affected. Let ℓ be the left sibling of x . SIBLINGCHAIN is preserved by setting ℓ 's right child to be r , and if r exists, setting r 's left sibling to ℓ . ■

27.4 Pairing Heap Methods

In this section we present internal and public methods for the PairingHeap class.

27.4.1 Constructors and Factory Methods

The constructor that takes no parameters creates an empty pairing heap that uses the default comparator.

```
public PairingHeap(){
    this(Objects.DEFAULT_COMPARATOR);
}
```

The constructor that takes *comp*, a user-provided comparator, creates an empty pairing heap that uses the given comparator.

```
public PairingHeap(Comparator<? super E> comp) {
    super(comp);
    root = null;
    FORE.setNext(AFT);
    FORE.markDeleted();
    AFT.markDeleted();
}
```

//satisfy Parent
//satisfy IterationList
//satisfy InUse

Correctness Highlights: SIZE is satisfied by the abstract collection constructor. Since this collection is empty, REACHABLE, HEAPORDERED, CHILDREN, PARENT, SIBLINGCHAIN, vacuously hold. Similarly, REDIRECTCHAIN holds since there are no removed nodes. ITERATIONLIST holds since the iteration order is FORE followed by AFT which is consistent with having *FORE.next = AFT* and *AFT.prev = FORE*.

By the correctness of the heap node *markDeleted* method, REMOVED holds since both *FORE* and *AFT* are marked as deleted and there are no other nodes. Finally, setting the root to *null* establishes PARENT. ■

The *reinitialize* method reinitializes all instance variables so this leftist heap is empty.

```
void reinitialize() {
    size = 0;
    root = null;
    FORE.setNext(AFT);
}
```

Correctness Highlights: Setting *size* to 0 preserves SIZE. By setting *root* to *null*, REACHABLE is preserved. The remaining properties hold vacuously since there are no elements in the heap. ■

The *newHeapNode* method takes *value*, the desired element, and returns a new heap node holding the given element.

```
HeapNode<E> newHeapNode(E value) {
    return new HeapNode<E>(value);
}
```

27.4.2 Algorithmic Accessors

The *max* method returns a highest priority element. It throws a *NoSuchElementException* when this pairing heap is empty.

```

public E max() {
    if (isEmpty())
        throw new NoSuchElementException();
    return root.element;
}

```

Correctness Highlights: Follows directly from `HEAPORDERED` since all descendants of the root have an equal or lower priority. ■

Although there is no efficient search method for a pairing heap, we include a linear time internal method to locate an element. This method uses the iteration list to visit each element until an equivalent element is found.

```

HeapNode<E> find(E element){
    HeapNode<E> ptr = FORE.next;                                //first element in iteration list
    while (ptr != AFT){
        if (equivalent(ptr.element, element))
            return ptr;
        ptr = ptr.next;
    }
    return null;
}

```

Correctness Highlights: By `ITERATIONLIST`, if `AFT` is reached then there is no equivalent element in the collection. ■

The method *getEquivalentElement* takes *target*, the target element, and returns an equivalent element. It throws a *NoSuchElementException* when there is no equivalent element in this collection.

```

public E getEquivalentElement(E target) {
    HeapNode<E> node = find(target);
    if (node != null)
        return node.element;
    else
        throw new NoSuchElementException();
}

```

Correctness Highlights: Follows from the correctness of *find*. ■

27.4.3 Representation Mutators

Similar to a leftist heap, all mutating methods of a pairing heap use *merge*, which takes *rootA*, the root of one pairing heap to merge, and *rootB*, the root of the other pairing heap to merge, and returns a reference to the root of the merged pairing heap. This method requires that $T(\text{rootA})$ and $T(\text{rootB})$ are valid pairing heaps. More specifically $T(\text{rootA})$ and $T(\text{rootB})$ must satisfy `HEAP-ORDERED`, `CHILDREN`, `PARENT`, `SIBLINGCHAIN`, and `REMOVED`. This method guarantees that

the resulting heap includes each element from $T(\text{rootA})$ and $T(\text{rootB})$ exactly once. It also preserves HEAPORDERED, CHILDREN, PARENT, SIBLINGCHAIN, and REMOVED.

```

HeapNode<E> merge(HeapNode<E> rootA, HeapNode<E> rootB) {
    if (comp.compare(rootA.element, rootB.element) > 0) {
        rootA.addChild(rootB);
        return rootA;
    } else {
        rootB.addChild(rootA);
        return rootB;
    }
}

```

Correctness Highlights: It is easily seen that HEAPORDERED and REACHABLE are preserved. By the correctness of *addChild*, CHILDREN, and PARENT are preserved. The other properties are not affected. ■

27.4.4 Content Mutators

Methods to Perform Insertion

As with the leftist heap, insertion into a pairing heap is performed by creating a pairing heap with the new element, and then using *merge* to combine this singleton pairing heap with the existing pairing heap. When the new element has a higher priority than all existing elements, it becomes the new root, and the existing pairing heap becomes its only child. A sequence of insertions is shown in [Figure 27.3](#).

The *mergeWithRoot* method takes x , a reference to the root of the pairing heap $T(x)$ that is to be merged with $T(r)$. It requires that x is not *null*.

```

void mergeWithRoot(HeapNode<E> x) {
    if (root == null) //special case for an empty collection
        root = x;
    else
        root = merge(root, x);
}

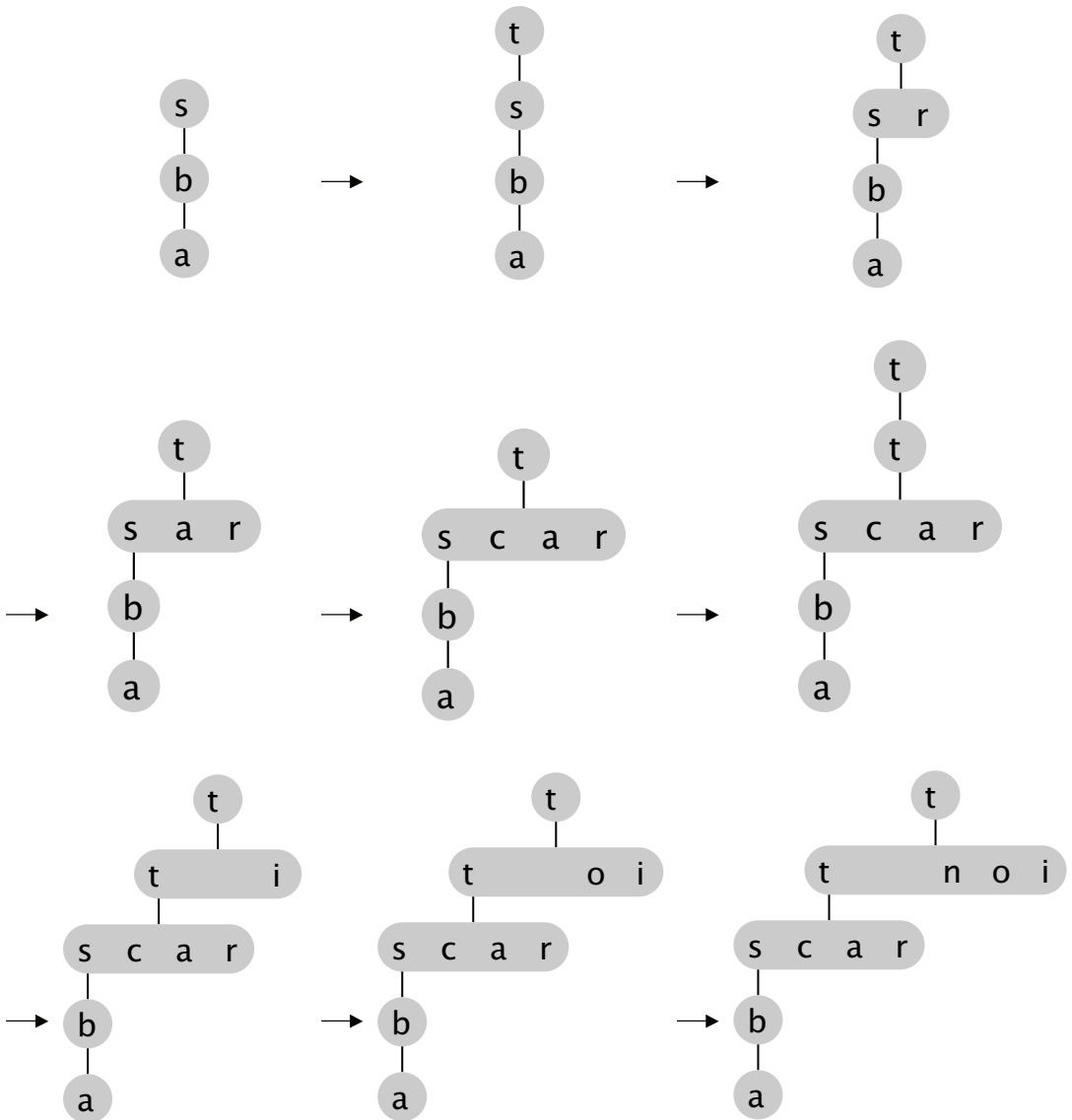
```

The internal *insert* method takes *element*, the new element to insert into this collection. It returns a reference to the heap node holding the new element.

```

HeapNode<E> insert(E element) {
    HeapNode<E> x = new HeapNode(element);
    mergeWithRoot(x);
    AFT.prev.setNext(x);
    x.setNext(AFT); //preserve IterationList by
    size++;         //adding new node just before AFT
    return x;       //preserve Size
}

```

**Figure 27.3**

The sequence of pairing heaps that results when inserting the letters “abstraction,” in that order, into an initially empty pairing heap. Whenever the new element is larger than all other elements in the collection, it becomes the new root with the current pairing heap as its only child. We start by showing the pairing heap obtained after inserting “abs.”

Correctness Highlights: By REACHABLE, when the root is *null*, the collection is empty. Clearly, in this case, setting the root to a node with the new element preserves all properties except SIZE and ITERATIONLIST. Likewise, when the collection is not empty, by the correctness of *merge*, all properties except SIZE and ITERATIONLIST are preserved. The node is correctly inserted just before AFT to preserve ITERATIONLIST. Finally, incrementing the instance variable *size* preserves SIZE. ■

The *add* method takes *element*, the element to add, and adds it to the heap.

```
public void add(E element) {
    insert(element);
}
```

Correctness Highlights: Follows from the correctness of *insert*. ■

The *addTracked* method takes as its parameter *element*, the element to add. It returns a tracker to the newly added element.

```
public PriorityQueueLocator<E> addTracked(E element) {
    HeapNode<E> node = insert(element);
    Tracker t = new Tracker(node);
    return (PriorityQueueLocator<E>) t;
}
```

Correctness Highlights: Follows from the correctness of *insert* and the Tracker constructor. ■

Methods to Modify Elements

Most pairing heap mutators rely heavily on *merge*. Whether the aim is to update a nodes value or remove it, the basic approach starts by detaching a portion of the pairing heap to form a “new” pairing heap. As in a leftist heap, when an element is replaced by a lower priority element, its children are detached and then merged back into the original heap. However, unlike a leftist heap in which each node as at most two children, each pairing heap node can have any number of children. The solution is a method that moves all of the children of a given node into a queue, and another method that merges the pairing heaps from the queue into the remainder of the original pairing heap. For example, consider the task of decreasing the priority of node *x*, where *x* has children $x_1, \dots, x_k$. First $x_1, \dots, x_k$ are detached from *x* and placed into a queue, enabling *x*’s priority to be decreased without violating HEAPORDERED. Then $T(x_1), \dots, T(x_k)$ are merged into what remains of $T(r)$. One might imagine merging the detached trees one by one. However, a substantial efficiency gain is realized by merging them pairwise, melding them into one heap that is finally merged with the root.

The *moveChildrenToQueue* method takes *x*, a reference to the node whose children are to be moved into the queue. It returns a queue of heap nodes corresponding to the children of the given node. When node *x* is placed in the queue, conceptually, this corresponds to placing $T(x)$ into the queue. This method requires that *x* is not *null*.

```
Queue<HeapNode<E>> moveChildrenToQueue(HeapNode<E> x){
    HeapNode<E> ptr = x.child;
    x.child = null;
    while (ptr != null) {
        HeapNode<E> right = ptr.sibR;
```

//detach children from node
//traverse through sibling chain
//remember the right sibling

```

        ptr.makeRoot();                                //make a root
        detached.enqueue(ptr);                          //put in queue
        ptr = right;                                    //move to next sibling
    }
    return detached;
}

```

Correctness Highlights: By CHILD, before the while loop begins, *ptr* references the leftmost child of *x*. Setting *x.child* to *null*, detaches all of *x*'s children from *x*. By SIBLINGCHAIN, the while loop will visit each sibling exactly once, and terminate when the rightmost sibling is reached. The rest of the correctness follows from that of *makeRoot* and the correctness of the queue implementation.

The internal *mergeQueue* method returns a reference to the root of the pairing heap that is the union of all pairing heaps in the queue of detached heaps. Merges are done pairwise, with each newly merged heap added to the back of the queue *detached* until there is only one heap in the queue. Observe that even if the queue has size *n*, only $\lceil \log_2 n \rceil$ merges are required.

```

HeapNode<E> mergeQueue() {
    while (detached.getSize() > 1)
        detached.enqueue(merge(detached.dequeue(), detached.dequeue()));
    return detached.dequeue();
}

```

Correctness Highlights: Correctness follows from the *merge* method and the correctness of the queue implementation. ■

The method *decreasePriority* takes *x*, a reference to the pairing heap node, and *element*, the new element to replace *x.element*. This method requires *x* is not *null*, and that *element* $\leq$ *x.element*. The approach taken in this method is to detach all children, merge them to form a single pairing heap, which is then merged with what remains in *T(root)*. This method could be optimized by modifying *moveChildrenToQueue* to move a node into the queue only if it would cause a violation with HEAPORDERED.

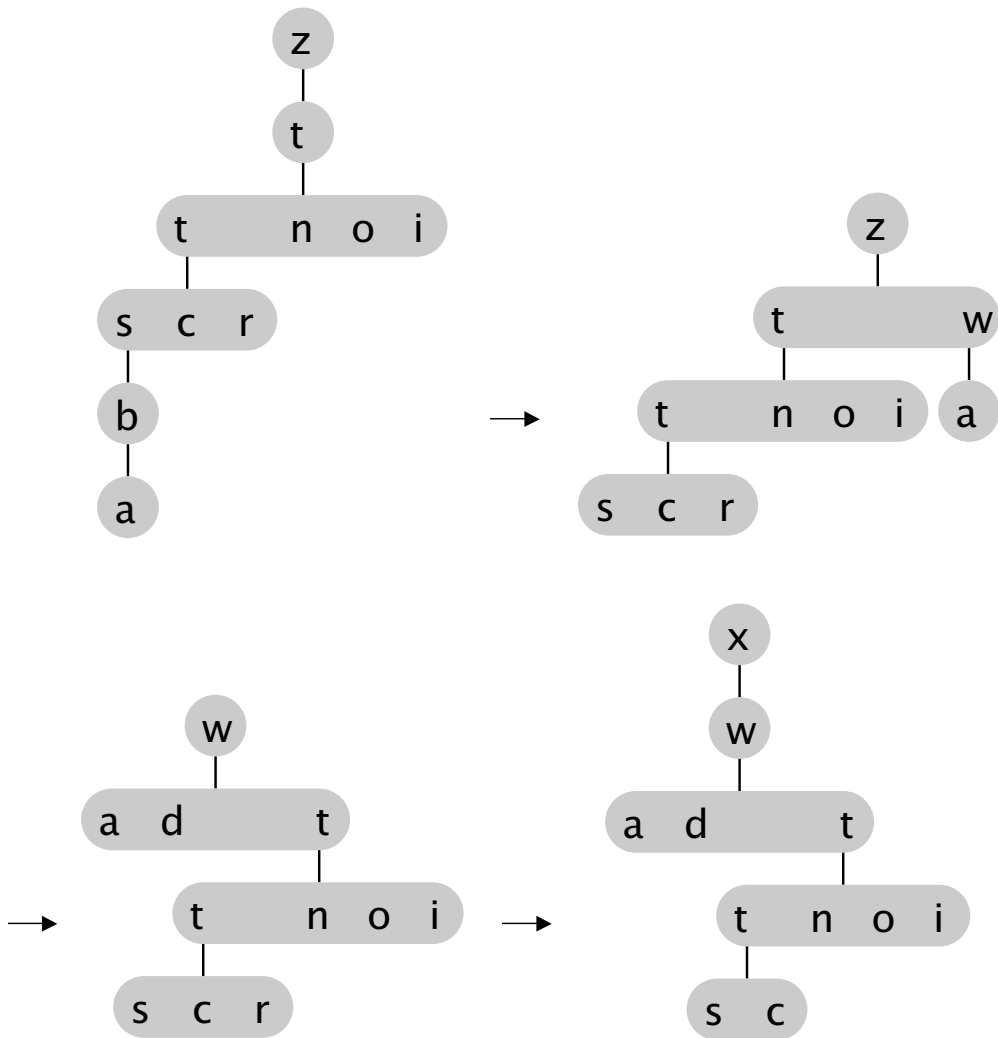
```

void decreasePriority(HeapNode<E> x, E element) {
    x.element = element;
    if (x.child != null) {                                //if node has a child
        moveChildrenToQueue(x);
        HeapNode<E> melded = mergeQueue();
        root = merge(root, melded);
        root.makeRoot();                                //preserve Parent for the root
    }
}

```

Correctness Highlights: Observe that once all children are removed from *node*, lowering its priority will not violate HEAPORDERED. The rest of the correctness follows from that of *moveChildrenToQueue*, *mergeQueue*, *merge*, and *makeRoot*. ■

The *increasePriority* method takes *x*, a reference to the heap node and *element*, the new element to replace *x.element*. This method requires that *x* is not *null*, *element* $\geq$ *x.element*. The approach is to detach *x* from its parent, and then use *merge* is used to recombine the detached heap with *T(root)*. While it would seem natural to only detach *x* from its parent if HEAPORDERED is violated, the only way to reach the parent would be to traverse the sibling chain, which could be long. Thus, *x* is detached without comparing to the parent.

**Figure 27.4**

An illustration of several updates (through a tracker) for the PairingHeap shown in [Figure 27.3](#). To reach the first diagram, one of the “a”s is changed to “z.” Next “b” is changed to “w.” Then the “z” is changed to a “d,” and finally “r” is changed to an “x.”

```

void increasePriority(HeapNode<E> x, E element){
    x.element = element;
    if (x != root){                                     //if node is not the root
        x.removeFromChain();
        x.sibL = x.sibR = null;
        root = merge(root, x);
        root.makeRoot();                               //preserve Parent for the root
    }
}

```

Finally, the internal *update* method takes *x*, a reference a node holding the element to be replaced, and *element*, the replacement element. This method requires that *x* is not *null*. Figure 27.4 illustrates a sequence of updates.

```

protected void update(HeapNode<E> x, E element){
    if (comp.compare(element, x.element) > 0)
        increasePriority(x, element);
    else if (comp.compare(element, x.element) < 0)
        decreasePriority(x, element);
    else
        x.element = element;
}

```

Correctness Highlights: When the new element has the same priority as the current element, then HEAPOORDERED is preserved. The rest of the correctness follows from that of *increasePriority* and *decreasePriority*. ■

Methods to Perform Deletion

The internal *remove* method takes *x*, a reference to the heap node to remove, and removes the node from the heap. It returns the element held in the removed node. This method requires that the given node is not *null*, and that it is not already deleted. If the node being removed has no children, it can just be removed from its sibling chain. Otherwise, all children are combined, via a sequence of merges performed by *mergeQueue*, into a single pairing heap *melded*. If the removed element was the root, then *root* can be set to *melded*. Otherwise, what remains in $T(\text{root})$ is merged with *melded*.

```

E remove(HeapNode<E> x) {
    if (x.child == null) {                               //when x has no children
        x.removeFromChain();                             //just remove from its sibling list
    }
    else {                                               //x has at least one child
        moveChildrenToQueue(x);
        HeapNode<E> melded = mergeQueue();
        if (x == root)
            root = melded;
        else {
            x.removeFromChain();
            root = merge(root, melded);
        }
    }
}

```

```

    }
}
x.markDeleted();
x.prev.setNext(x.next);
size--;
return x.element;
}
//preserve Removed
//preserve IterationList
//preserve Size

```

Correctness Highlights: Since all nodes in $T(\text{root})$ except for *node*, are placed back into $T(\text{root})$, REACHABLE is preserved. By the correctness of the internal methods used, HEAPORDERED is preserved. By the correctness of *removeFromList*, *moveChildrenToQueue*, *mergeQueue*, and *merge*, CHILDREN, PARENT, and SIBLINGCHAIN are preserved. By the correctness of the heap node *markDeleted* method, REMOVED and REDIRECTCHAIN are preserved. Finally, the last two lines preserve ITERATIONLIST and SIZE. ■

The public *remove* method takes *element*, the target. It removes an equivalent element from the collection. It returns *true* if and only if an equivalent element was found, and hence removed.

```

public boolean remove(E element) {
    HeapNode<E> node = find(element);
    if (node != null) {
        remove(node);
        return true;
    }
    return false;
}

```

The *extractMax* method returns and removes the highest priority element from the collection. It throws a *NoSuchElementException* when the collection is empty.

```

public E extractMax() {
    if (isEmpty())
        throw new NoSuchElementException();
    return remove(root);
}

```

Correctness Highlights: Follows from HEAPORDERED and the *remove* method. ■

27.4.5 Locator Initializers

The *iterator* method creates a new tracker that is at FORE.

```

public PriorityQueueLocator<E> iterator() {
    return new Tracker(FORE);
}

```

The method *getLocator* takes *element*, the target. It returns a tracker positioned at the given element. It throws a *NoSuchElementException* when there is no equivalent element in this collection.

```

public PriorityQueueLocator<E> getLocator(E element) {
    HeapNode node = find(element);
    if (node == null)
        throw new NoSuchElementException();
    else
        return new Tracker(find(element));
}

```

27.5 Tracker Inner Class

AbstractCollection<E>.AbstractLocator<E> implements *Locator*<E>

↑ **Tracker** implements *PriorityQueueLocator*<E>

Each tracker references the heap node it tracks.

```

HeapNode<E> node;

```

The constructor takes *node*, a reference to the node to track.

```

Tracker(HeapNode<E> node) {
    this.node = node;
}

```

Recall that *inCollection* returns *true* if and only if the tracked element is currently in the collection.

```

public boolean inCollection() {
    return !node.isDeleted();
}

```

The *get* method returns the tracked element. It throws a *NoSuchElementException* when tracker is not at an element in the collection.

```

public E get() {
    if (node.isDeleted())
        throw new NoSuchElementException();
    return node.element;
}

```

The *skipRemovedElements* method takes *ptr*, a pointer to a list item, and returns a pointer to the first reachable list item (including *ptr* itself if it is reachable) obtained by following the *next* pointer of any unreachable list item. More formally, $x_{pos}(ptr)$ is returned. This method performs compression of the chain of deleted elements.

```

private HeapNode<E> skipRemovedElements(HeapNode<E> ptr) {
    if (!ptr.isDeleted() || ptr == AFT)
        return ptr;
    ptr.next = skipRemovedElements(ptr.next);
    return ptr.next;
}

```

Correctness Highlights: We proceed by induction on the number of recursive calls made by *skipRemovedElements*. If *ptr* is not at a removed node or is at AFT, then correct value is returned, and since no updates occur, all properties are preserved. We now consider the inductive step. For a heap node *ptr* that is not in use, by definition $\text{succ}(\text{ptr}) = \text{succ}(\text{ptr.next})$. Thus the correct value is returned, and the update to *ptr.next* preserves REMOVED.

As an optimization, every unreachable heap node *y* accessed will be updated to reference the reachable heap node $\text{succ}(y)$, to compress the chain. ■

The *advance* method moves the tracker to the next element in the iteration order (or AFT if the last element is currently tracked). It returns *true* if and only if after the update, the tracker is at an element of the collection. It throws an *AtBoundaryException* when the tracker is at AFT since there is no place to advance.

```
public boolean advance() {
    checkValidity();
    if (node == AFT)
        throw new AtBoundaryException("Already after end.");
    if (node.isDeleted()) //if tracked element has been deleted
        node = skipRemovedElements(node); //update tracker to successor
    else
        node = node.next;
    return node != AFT;
}
```

Correctness Highlights: If the tracker is currently at AFT, then the required exception is thrown. By the correctness of *skipRemovedElements*, when the tracked element is not in the collection, *ptr* is updated to track the next element in the iteration order.

Next we consider the case when the tracked element is in the collection. In this case, *ptr* is advanced to the next element in the collection. In both cases, *true* is correctly returned as long as the updated tracker location is not AFT. ■

The *retreat* method moves the tracker to the previous element in the iteration order (or FORE if the first element is currently tracked). It returns *true* if and only if after the update, the tracker is at an element of the collection. It throws an *AtBoundaryException* when the tracker is at FORE since there is no place to retreat.

```
public boolean retreat() {
    if (node == FORE)
        throw new AtBoundaryException("Already before start.");
    if (node.isDeleted()) //if tracked element has been deleted
        node = skipRemovedElements(node); //update tracker to successor
    node = node.prev;
    return node != FORE;
}
```

Correctness Highlights: If the tracker is currently at FORE, then the required exception is thrown. We first consider when *skipRemovedElements* is called. By REMOVED, this situation occurs exactly when the tracked node had been removed from the collection. The rest of the correctness follows from that of *skipRemovedElements* and ITERATIONLIST. ■

The *hasNext* method returns *true* if there is some element after the current tracker position.

```
public boolean hasNext() {
    return ((node.next != AFT) &&
        (skipRemovedElements(node) != AFT));
}
```

Correctness Highlights: Follows from ITERATIONLIST and the correctness of *skipRemovedElements*. ■

As discussed in Section 5.8, the *remove* method removes the tracked element and updates the tracker to be at the element in the iteration order that preceded the one removed. It throws a *NoSuchElementException* when the tracker is at FORE or AFT.

```
public void remove() {
    if (node.isDeleted())
        throw new NoSuchElementException();
    PairingHeap.this.remove(node);
}
```

The *update* method replaces the tracked element by *element*. It requires *element* is different than the element at the current tracker position.

```
public void update(E element) {
    if (node.isDeleted())
        throw new NoSuchElementException();
    PairingHeap.this.update(node, element);
}
```

The *increasePriority* method replaces the tracked element by *element*. It requires that the given parameter is greater than *e*, or that *e* is the parameter being passed and its value has been mutated to have a higher priority than it had previously. That is, it is acceptable practice to mutate the element to have a higher priority and then immediately call *increasePriority* to restore the properties of the priority queue.

```
public void increasePriority(E element) {
    if (node.isDeleted())
        throw new NoSuchElementException();
    PairingHeap.this.increasePriority(node, element);
}
```

The *decreasePriority* method replaces the tracked element by *element*. It requires that the given parameter is less than *e*, or that *e* is the parameter being passed and its value has been mutated to have a lower priority than it had previously. That is, it is acceptable practice to mutate the element to have a lower priority and then immediately call *decreasePriority* to restore the properties of the priority queue.

```
public void decreasePriority(E element) {
    if (node.isDeleted())
        throw new NoSuchElementException();
    PairingHeap.this.decreasePriority(node, element);
}
```


method	amortized time complexity	worst-case time complexity
<i>ensureCapacity(x)</i> <i>max()</i>	$O(1)$ $O(1)$	$O(1)$ $O(1)$
<i>add(o)</i> <i>addTracked(o)</i>	$O(\log n)$ $O(\log n)$	$O(1)$ $O(1)$
<i>extractMax()</i>	$O(\log n)$	$O(n)$
<i>clear()</i> <i>contains(o)</i> <i>getEquivalentElement(o)</i> <i>getLocator(o)</i> <i>remove(o)</i> <i>toArray()</i> <i>toString()</i> <i>trimToSize()</i> <i>accept()</i>	$O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$	$O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$
<i>addAll(c)</i>	$O(c \log n)$	$O(n)$
<i>retainAll(c)</i>	$O(c n)$	$O(c n)$

Table 27.5 Summary of the asymptotic time complexities for the *PriorityQueue* public methods when implemented using a pairing heap.

27.6 Performance Analysis

The asymptotic time complexities of all public methods for the *PairingHeap* class are shown in Table 27.5, and the asymptotic time complexities for all of the public methods of the *PairingHeapTracker* class are given in Table 27.6.

Observe that the space complexity requires allocating one heap node for every element. For a threaded pairing heap, as implemented here, there are six instance variables per node. So the space usage is roughly that of $6n$ references. (In an unthreaded implementation, there would be only four instance variables per node.)

The amortized analysis is quite complex, so we do not cover it here. Readers interested can find this analysis in the recent paper of Pettie [124] that discusses the original analysis of Fredman et al. [64] and provides an improved analysis.

We now analyze the worst-case time complexities. Since the tracker constructor takes constant time, so does *iterator*. Also, since the pairing heap is an elastic implementation, *ensureCapacity* and *trimToSize* do not perform any computation, so they take constant time.

The highest priority element is always in the root, so *max* takes constant time. For all other elements, while *find* performs an optimization to minimize the costs, the target element may need to be compared to all elements in the collection. Thus, in the worst-case, *find*, *getEquivalentElement*, and *getLocator* take linear time.

The *merge* method takes constant time since it just compares the two roots, making the lower priority one the parent of the other. Since *add* and *addTracked* just allocate and initialize the new node, merge it with $T(\text{root})$, and place it in at the end of the iteration list, they have constant worst-case cost. Also, observe that once a node is located, *increasePriority* detaches the node, changes the element, and then uses a single merge to combine the removed subtree with the remaining portion

method	amortized time complexity	worst-case time complexity
<i>get()</i>	$O(1)$	$O(1)$
<i>advance()</i>	$O(1)$	$O(n)$
<i>hasNext()</i>	$O(1)$	$O(n)$
<i>next()</i>	$O(1)$	$O(n)$
<i>retreat()</i>	$O(1)$	$O(n)$
<i>update(o)</i> , <i>increase priority</i>	$O(\log n)$	$O(1)$
<i>remove()</i>	$O(\log n)$	$O(n)$
<i>update(o)</i> , <i>decrease priority</i>	$O(\log n)$	$O(n)$

Table 27.6 Summary of the time complexities for the locator methods of the pairing heap tracker.

of the pairing heap. Thus, it takes constant time.

The cost for *moveChildrenToQueue* is proportional to the number of children since *enqueue* takes constant time. Likewise, since *merge* takes constant time, *mergeQueue* takes time logarithmic in the size of the queue since pairwise merges are performed. After a node to remove or update has been located, the process of removing the node or decreasing its priority is dominated by the cost of placing all its children in a queue that is then merged with the remaining portion of $T(\text{root})$. Thus these methods both have worst-case cost proportional to the number of children, which can be $O(n)$ in the worst case.

The *clear*, *toString*, *toArray*, and *accept* methods use the iteration list, so they take linear time. The *addAll* method takes constant time per element, leading to an overall cost of $O(|c|)$.

The *retainAll* method performs a search for each element in the provided collection c . Any element not found in c is removed. In the worst-case, for each element a linear time search must be performed, and in the worst-case it takes linear time to remove it from the pairing heap. So the worst-case cost is $O(|c| \cdot n)$.

We now analyze the methods in the Tracker class. The *get*, *inCollection*, and *set* take constant time since they only require access to the heap node that is referenced by the tracker. Observe that each heap node can only be reached once via a call to *skipRemovedElements* that does not begin at that node. Thus the time required by *skipRemovedElements* is proportional to the number of removed elements that are accessed before reaching an element in the collection. In the worst-case it could have linear cost. However, since this method modifies all references followed to directly reference an element in the collection, the cost of this method is constant when amortized over *remove* calls made for the collection.

When the tracker is at an element in the collection, *advance*, *hasNext*, *next*, and *retreat* take constant time since they can use the iteration list. Thus, their time is dominated by *skipRemovedElements*. So while these methods have worst-case linear time, their amortized cost is constant. Also, the worst-case cost would only be high if a long sequence of adjacent elements were removed.

27.7 Quick Method Reference

PairingHeap Public Methods

- p. 406** *PairingHeap*()
- p. 406** *PairingHeap*(Comparator(? super E) comp)

- p. 98 void **accept**(Visitor<? super E> v)
- p. 410** void **add**(E element)
- p. 100 void **addAll**(Collection<? extends E> c)
- p. 410** PriorityQueueLocator<E> **addTracked**(E element)
- p. 100 void **clear**()
- p. 97 boolean **contains**(E value)
- p. 99 void **ensureCapacity**(int capacity)
- p. 414** E **extractMax**()
- p. 96 int **getCapacity**()
- p. 96 Comparator<? super E> **getComparator**()
- p. 98 E **getEquivalentElement** (E target)
- p. 414** PriorityQueueLocator<E> **getLocator**(E element)
- p. 96 int **getSize**()
- p. 96 boolean **isEmpty**()
- p. 414** PriorityQueueLocator<E> **iterator**()
- p. 406** E **max**()
- p. 414** boolean **remove**(E element)
- p. 100 void **retainAll**(Collection<E> c)
- p. 97 Object[] **toArray**()
- p. 97 E[] **toArray**(E[] array)
- p. 98 String **toString**()
- p. 99 void **trimToSize**()

PairingHeap Internal Methods

- p. 97 int **compare**(E e1, E e2)
- p. 411** void **decreasePriority**(HeapNode<E> x, E element)
- p. 97 boolean **equivalent**(E e1, E e2)
- p. 407** HeapNode<E> **find**(E element)
- p. 411** void **increasePriority**(HeapNode<E> x, E element)
- p. 408** HeapNode<E> **insert**(E element)
- p. 408** HeapNode<E> **merge**(HeapNode<E> rootA, HeapNode<E> rootB)
- p. 411** HeapNode<E> **mergeQueue**()
- p. 408** void **mergeWithRoot**(HeapNode<E> x)
- p. 410** Queue<HeapNode<E>> **moveChildrenToQueue**(HeapNode<E> x)
- p. 406** HeapNode<E> **newHeapNode**(E value)
- p. 406** void **reinitialize**()
- p. 413** E **remove**(HeapNode<E> x)
- p. 99 void **traverseForVisitor**(Visitor<? super E> v)
- p. 413** void **update**(HeapNode<E> x, E element)
- p. 98 void **writeElements**(StringBuilder s)

PairingHeap.HeapNode Public Methods

- p. 403** *HeapNode*(E element)

PairingHeap.HeapNode Internal Methods

- p. 404** void **addChild**(HeapNode<E> newChild)
- p. 404** boolean **isDeleted**()
- p. 403** boolean **isLeftmostChild**()
- p. 404** boolean **isRightmostChild**()
- p. 403** void **makeRoot**()
- p. 404** void **markDeleted**()
- p. 405** void **removeFromChain**()
- p. 403** void **setLeft**(HeapNode<E> ptr)
- p. 403** void **setNext**(HeapNode<E> ptr)

PairingHeap.Tracker Public Methods

- p. 416** boolean **advance**()
- p. 417** void **decreasePriority**(E element)
- p. 415** E **get**()
- p. 417** boolean **hasNext**()
- p. 101 void **ignoreConcurrentModifications**(boolean ignore)
- p. 101 void **ignorePriorConcurrentModifications**()
- p. 415** boolean **inCollection**()
- p. 417** void **increasePriority**(E element)
- p. 101 E **next**()
- p. 417** void **remove**()
- p. 416** boolean **retreat**()
- p. 417** void **update**(E element)

PairingHeap.Tracker Internal Methods

- p. 415** *Tracker*(HeapNode<E> node)
- p. 101 void **checkValidity**()
- p. 415** HeapNode<E> **skipRemovedElements**(HeapNode<E> ptr)
- p. 101 void **updateVersion**()

Chapter 28

Fibonacci Heap Data Structure

AbstractCollection<E> implements *Collection*<E>

└─ PairingHeap<E> implements *PriorityQueue*<E>, *Tracked*<E>

└─ **FibonacciHeap**<E> implements *PriorityQueue*<E>, *Tracked*<E>

Uses: Java references

Used By: TaggedFibonacciHeap (Section 49.8.5)

Strengths: The Fibonacci heap is theoretically the best data structure. It is the only priority queue data structure with constant amortized cost for merging two priority queues, and also increasing the priority of an element through a locator. The pairing heap, in contrast, has a logarithmic amortized cost for both of these operations. Since Prim's minimum spanning tree algorithm and Dijkstra's shortest path algorithm are dominated by the cost of increasing the priority of elements, the Fibonacci heap yields the theoretically best worst-case time complexities for these algorithms.

Weaknesses: While the Fibonacci heap is theoretically best, experimental studies [142, 104] have shown that, in practice, the pairing heap outperforms the Fibonacci heap. In particular, the experiments of Moret and Shapiro [116] led to the conclusion that using a pairing heap within Prim's minimum spanning algorithm leads to a more efficient solution than using a Fibonacci heap, or a variety of other priority queue data structures. Another drawback, is that the Fibonacci heap is a fairly complex data structure to implement, as compared to the binary heap, leftist heap, and pairing heap.

Critical Mutators: Since our implementation is threaded, there are no critical mutators. In an unthreaded implementation *add*, *addAll*, *addTracked*, *extractMax*, *remove*, and *retainAll* would be critical mutators.

Competing Data Structures: Unless theoretical bounds a Fibonacci heap provides are essential, a pairing heap (Chapter 27) should be considered. If there is a need to efficiently merge data structures, yet increasing the priority of elements is relatively rare, then a leftist heap (Chapter 26) is a good option. Finally, the binary heap (Chapter 25) is much more space efficient, and generally a good competitor if there is no need to efficiently merge two priority queues, and increasing the priority of elements is relatively rare.

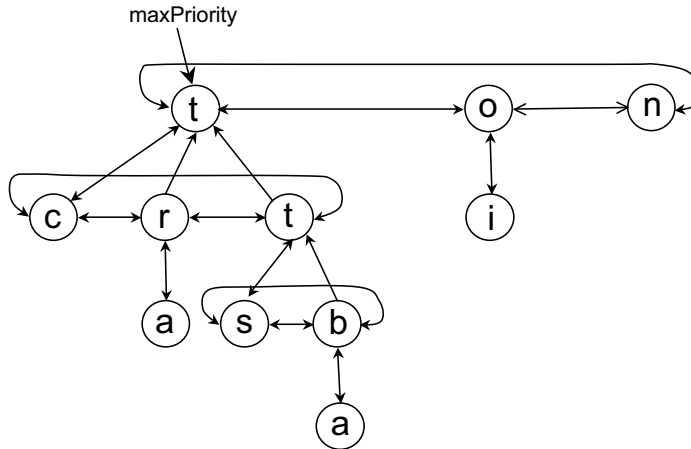


Figure 28.1

The internal representation for a populated example for a Fibonacci heap holding the letters of “abstraction.” Horizontal arrays show the *left* and *right* sibling references and the other arrows show the *child* (downward) and *parent* (upward) references.

28.1 Internal Representation

A Fibonacci heap is an unordered forest of **heap-ordered trees**, which means that although the trees themselves are heap-ordered, there is no specified ordering among the trees. In contrast, a binomial heap [153], a precursor to the Fibonacci heap, is an ordered forest of heap-ordered trees. More specifically, the trees in the forest are ordered by the number of children of their roots.

Like a pairing heap, the children of a Fibonacci heap node are stored as a sibling chain. However, unlike a pairing heap, in a Fibonacci heap each child has a reference to its parent, and the sibling chain is a circular doubly linked list. The roots for all of the trees in the forest are store in a root chain, which has a structure like that of the sibling chain. Each Fibonacci heap maintains a reference to the highest priority element in the root chain. Since the trees are heap-ordered, it follows that this node holds a highest priority element.

The other important changes, which add to the code complexity but help support the low amortized costs, are that each heap node maintains a **degree**, which is the number of children it has, and a **mark** bit to record whether the node has had a child removed since the last time its parent changed.

Instance Variables and Constants: All variables are inherited from the PairingHeap class.

Populated Example: Figure 28.1 shows the internal representation for a Fibonacci heap (excluding the iteration list) holding the letters of “abstraction.” It was obtained by inserting the letters in the word “abstraction” and then a “z” followed by a *extractMin*. The root chain is shown at the top. It uses the *left* and *right* references that are typically used for the sibling list, to instead form a **root chain**. Observe that the root chain and sibling lists are circular, doubly-linked list. A root node is distinguished by being its own parent.

Throughout this chapter we use a more compact visualization of a Fibonacci heap, which shows all siblings together in a shaded rounded rectangle. The rounded rectangle at the top shows the root chain. We use the convention that *root* is always shown first in the root chain. As in the pairing

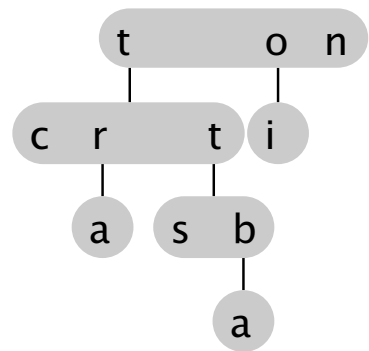


Figure 28.2
Our visual representation for the Fibonacci heap shown in Figure 28.1. This representation is the same as the one used for a Pairing heap.

heap visualization, the children of each element are connected to it by a line. Figure 28.2 shows the Fibonacci heap from Figure 28.1 in this form. The order for the iteration list, which is based on the order in which the elements were added to the collection, is completely independent of the structure of the tree. The iteration list is not shown in the figures.

Terminology: In addition to using some terminology defined in Chapter 27 for the pairing heap, we introduce the following new and modified terms.

- The **root chain** is a circular doubly-linked list that is referenced by *root*. Furthermore, we guarantee that *root* references a maximum priority element.
- The **sibling chain** is a circular doubly linked list, using *left* and *right* references, containing all children of a common parent.
- The **degree** of a node is the length of its sibling chain.

Abstraction Function: The abstraction function for the Fibonacci heap is the same as that for a pairing heap. We repeat it here for easy reference. Let *FH* be a Fibonacci heap instance. The abstraction function is

$$AF(FH) = seq(FORE.next).$$

Optimizations: As with the leftist heap and pairing heap, there is a trade-off between providing efficient support for iteration and minimizing the space usage. Our implementation of a Fibonacci heap provides fast and robust support for iteration at the cost of introducing $2n$ additional references. As illustrated for the leftist heap (Chapter 26), an alternative approach would be to use a visiting iterator to reduce the space complexity, at the cost of less efficient iteration.

A fundamental method of the Fibonacci heap is *consolidate*, which uses an auxiliary array with a slot for every possible degree. As proven in Section 28.5, for a Fibonacci heap of n nodes, all nodes have degree at most $d_n = \lceil \log_\phi n \rceil$ where $\phi = (1 + \sqrt{5})/2$. Since the possible degrees are $\{0, 1, \dots, d_n\}$, an array of size $d + 1$ is needed. To avoid allocating a new array each time *consolidate* is called, we allocate a static array that is reused. However, the size of the array is a function of n , which is changing. Using ideas similar to those introduced in for the dynamic array (Chapter 13), we resize this array as needed. Since the array size is logarithmic in n , it is resized infrequently.

28.2 Representation Properties

We inherit `SIZE` from `AbstractCollection`, and `REACHABLE`, `HEAPORDERED`, `ITERATIONLIST`, and `REMOVED` from `PairingHeap`. We modify `CHILDREN`, `PARENT`, and `SIBLINGCHAIN` to reflect the fundamental difference in the internal representation between the pairing heap and Fibonacci heap. We introduce four new properties. `DEGREE` and `MARKED` relate to the added instance variables. Since the Fibonacci heap is a forest, and not a tree, `HEAPORDERED` is no longer sufficient to guarantee that `root` references a maximum priority element. We introduce `ROOT` to provide this guarantee. Finally, `ROOTCHAIN` is like `SIBLINGCHAIN` but applies to the root chain only.

CHILDREN: For each reachable heap node x , $x.child = null$ if and only if x has no children.

DEGREE: For each reachable heap node x , $x.degree$ is always equal to the number of children of x .

PARENT: A reachable node x is in the root list if and only if $x.parent = null$. For each reachable heap node x that is not in the root chain, $x.parent.child$ refers to a sibling list containing x .

SIBLINGCHAIN: For each reachable heap node x not in the root chain, $x.right.left = x$.

ROOTCHAIN: For each heap node x in the root chain, $x.right.left = x$.

ROOT: The priority of the node referenced by `root` is at least as high as the priority of any node x in the root chain, and therefore holds highest priority element.

MARKED: For each reachable heap node x not in the root chain, $x.marked$ is `true` if and only if one and only one child has been removed from x since the last time x 's parent changed (or since it was created).

28.3 Fibonacci Heap Node Inner Class

PairingHeapNode<*E*>

↑ ***FibonacciHeapNode***<*E*>

In this section, we describe the `FibonacciHeapNode` inner class that extends the `PairingHeapNode`. Along with inheriting `element`, `child`, `sibL`, `sibR`, `next`, and `prev`, the following three instance variables are introduced. Also, note that `child` now references an arbitrary child, since there is no concept of a leftmost child in the circular sibling chain.

```
int degree = 0;                                //number of children (initially 0)
boolean marked = false;                        //true iff child detached since parent changed
FibonacciHeapNode<E> parent;                  //parent (null if in root chain)
```

The constructor takes `element`, the element to be held in this node.

```
public FibonacciHeapNode(E element) {
    super(element);
}
```


Correctness Highlights: The superclass constructor satisfies CHILDREN. Since a newly created node has no children, setting *degree* to 0 satisfies DEGREE. The other properties that apply to a heap node are satisfied by *makeRoot*. ■

The *makeRoot* method sets *sibL* and *sibR* so that this node becomes the root.

```
void makeRoot() {
    sibL = sibR = this;           //satisfies RootChain
    marked = false;              //parent changes when moved to root chain
    parent = null;               //satisfies Parent for a root
}
```

Correctness Highlights: By definition, a node should be unmarked when it is created, and whenever its parent changes. So initializing *marked* to *false* preserves MARKED. Setting *parent* to *null*, preserves ROOTCHAIN since the new node will be added to the root chain. Finally, *sibL* and *sibR* are initialized to preserve *RootChain* for the case in which the new node is the only root in the root chain. ■

The *addChild* method takes *newChild*, a reference to the heap node to be added as a child of this node. This method preserves CHILDREN, DEGREE, PARENT, MARKED, SIBLINGCHAIN, and ROOTCHAIN.

```
void addChild(HeapNode<E> newChild) {
    if (degree == 0) {           //newChild will be an only child
        child = newChild;        //satisfy Child
        newChild.sibR = newChild.sibL = newChild; //satisfy SiblingChain
    } else {                     //this node already has a child
        newChild.setLeft(child.sibL); //preserve SiblingChain
        child.setLeft(newChild);
    }
    ((FibonacciHeapNode<E>) newChild).parent = this; //preserve Parent
    ((FibonacciHeapNode<E>) newChild).marked = false; //preserve Marked
    degree++; //preserve Degree
}
```

Correctness Highlights: We use *x* to denote the node on which this method is called. We first consider when the *degree* of *x* is 0. By DEGREE, *x* has no children. Setting *child* to the new node satisfies CHILD, and having both the left and right sibling refers to the new child satisfies SIBLINGCHAIN.

Next we consider when *x* has at least one child. In this case, no change is required to preserve CHILDREN. The new node is introduced into the sibling chain (or root chain) just left of the node referenced by *child*, thus preserving SIBLINGCHAIN (and ROOTCHAIN).

Finally, in both cases the parent for the new node is set to preserve PARENT. Since *newChild* has a new parent, setting it to be unmarked, preserves MARKED. Finally, since *x* has been given a new child, incrementing *degree* preserves DEGREE. ■

The *removeFromChain* method removes this heap node from its current sibling chain. This method will preserve CHILDREN, DEGREE, PARENT, SIBLINGCHAIN, ROOTCHAIN, and MARKED.

```

void removeFromChain() {
    if (parent != null) {                                     //preserve Parent
        parent.degree--;                                     //preserve Degree
        if (parent.child == this) {
            if (this == sibR)
                parent.child = null;                         //it's an only child
            else
                parent.child = sibR;
        }
    }
    sibR.setLeft(sibL);                                     //Preserve SiblingChain (or RootChain)
    makeRoot();
}

```

Correctness Highlights: Let x be the node on which this method is called. If x has a parent, its parent will now have one less child, so decrementing the degree for its parent preserves DEGREE. If x is the child referenced by its parent, then CHILDREN must be preserved. By SIBLINGCHAIN, if x is its own right sibling (equivalently, the left sibling could have been used), then it is an only child. In that case, setting *parent.child* to *null* preserves CHILDREN. Otherwise, CHILDREN is preserved by setting *parent.child* to any other node in the sibling chain for x . In particular, x 's right sibling is used. Finally, setting x 's parent to *null* preserves PARENT.

If x has no parent, then it must be removed from the root chain, and if x has a parent, then it must be removed from its sibling chain. In both cases, by the correctness of *setLeft*, the sibling (or root) chain is correctly modified. Finally, MARKED is preserved by setting *marked* to *false*, since x 's parent has changed. ■

28.4 Fibonacci Heap Methods

In this section we present internal and public methods for the FibonacciHeap class.

28.4.1 Constructors and Factory Methods

The constructor that takes no parameters creates an empty Fibonacci heap that uses the default comparator.

```

public FibonacciHeap(){
    super(Objects.DEFAULT_COMPARATOR);
}

```

The constructor that takes *comp*, a user-provided comparator, creates an empty Fibonacci heap that uses the given comparator.

```

public FibonacciHeap(Comparator<? super E> comp) {
    super(comp);
}

```

The *newHeapNode* factory method takes *element*, the element to be stores in this heap node, and returns a new heap node holding the given element.

```
HeapNode<E> newHeapNode(E element) {
    return new FibonacciHeapNode<E>(element);
}
```

28.4.2 Representation Mutators

Unlike a leftist heap and a pairing heap, which rely heavily on the *merge* method to reorganize, a Fibonacci heap implements *merge* by just linking the two root chains together, and updating *root*. If no other method was used to modify the data structure, the result would be an unsorted doubly-linked circular list with a references to the highest priority element. The fundamental method used to reorganize a Fibonacci heap into a more efficient structure is *consolidate*. Prior to the execution of *consolidate*, *ROOT* may be temporarily violated. Along with reorganizing the Fibonacci heap, *consolidate* will also restore *ROOT*.

The goal of consolidating is to combine the trees by ensuring that no two nodes in the root chain have the same degree. An array *buckets* of references to heap nodes, is used to help with the bookkeeping. The slot *bucket[d]* will either reference a node in the root chain with degree *d* or be *null*. For ease of exposition, when a node *x* is referenced by *bucket[d]* we say that *x* is **in** bucket *d*. All nodes *x* in the root chain from left to right are processed using the following procedure. If *bucket[x.degree]* is empty, then *x* is placed in this bucket *d* to remember that it has degree *d*. However, if *bucket[x.degree]* holds node *y*, then *x* and *y* have the same degree. To ensure that the root chain has only one node with a given degree, *x* and *y* are **linked** by having the lower priority node among them become a child of the other. For ease of exposition, suppose *y* had a higher priority. Observe that by making *x* a child of *y*, the degree of *y* increases. At this point either *bucket[y.degree]* is *null*, in which case *y* is placed there, or otherwise the link process can be repeated until an empty bucket is reached. Then the same process is repeated for every node in the root chain, so eventually each node in the root chain is in its own bucket, and therefore has a unique value for degree.

Figure 28.3 illustrates the consolidation process as it occurs when *extractMax* is called on a Fibonacci heap. To begin the letters of “abstraction” have been added. We examine what happens when the leftmost “t” in the top diagram is removed. The nodes are processed from left to right. The “s” has degree 0, and is stored in bucket 0. Next, the “b” has degree 0, so bucket 0 is examined. Since “s” is already there, the “s” and “b” are linked. Since “s” is larger (i.e., has higher priority), “b” becomes its child. Thus now “s” has a degree of 1 and is stored in bucket 1. Bucket 0 is now empty. The left figure in the second row illustrates the Fibonacci heap at this point in the process.

Next “r” is processed and placed in bucket 0. When “a” is processed, it is linked with “r,” which becomes the parent of “a.” Since “r” now has degree 1, bucket 1 is examined and found to hold the tree rooted at “s.” Thus, these two trees are linked. Since “s” is larger than “r,” “s” becomes the root of this tree of degree 2 (holding s,b,r, and a). At this point “s” is moved to bucket 2. The Fibonacci heap at this point in the process is shown at the left of the third row.

Now the second “a” (which has degree 0) is processed and placed in bucket 0. Then “c” is processed and linked with “a” and placed in bucket 1. (This stage is shown in the right of the third row.) Next “t” and “i” are processed and linked as illustrated at the left of the fourth row of the figure. Since bucket 1 already holds “c,” the tree rooted at “c” (of degree 1) and the tree rooted at “t” (of degree 1) are linked creating a tree of degree 2 rooted at “t.” Next, bucket 2 is examined and found to hold the tree rooted at “s.” Thus these two trees are combined by making “s” a child of “t.” The figure on the bottom left shows the resulting tree of degree 3 (which is placed in bucket 3). Finally the “o” and “n” are merged to complete *consolidate*.

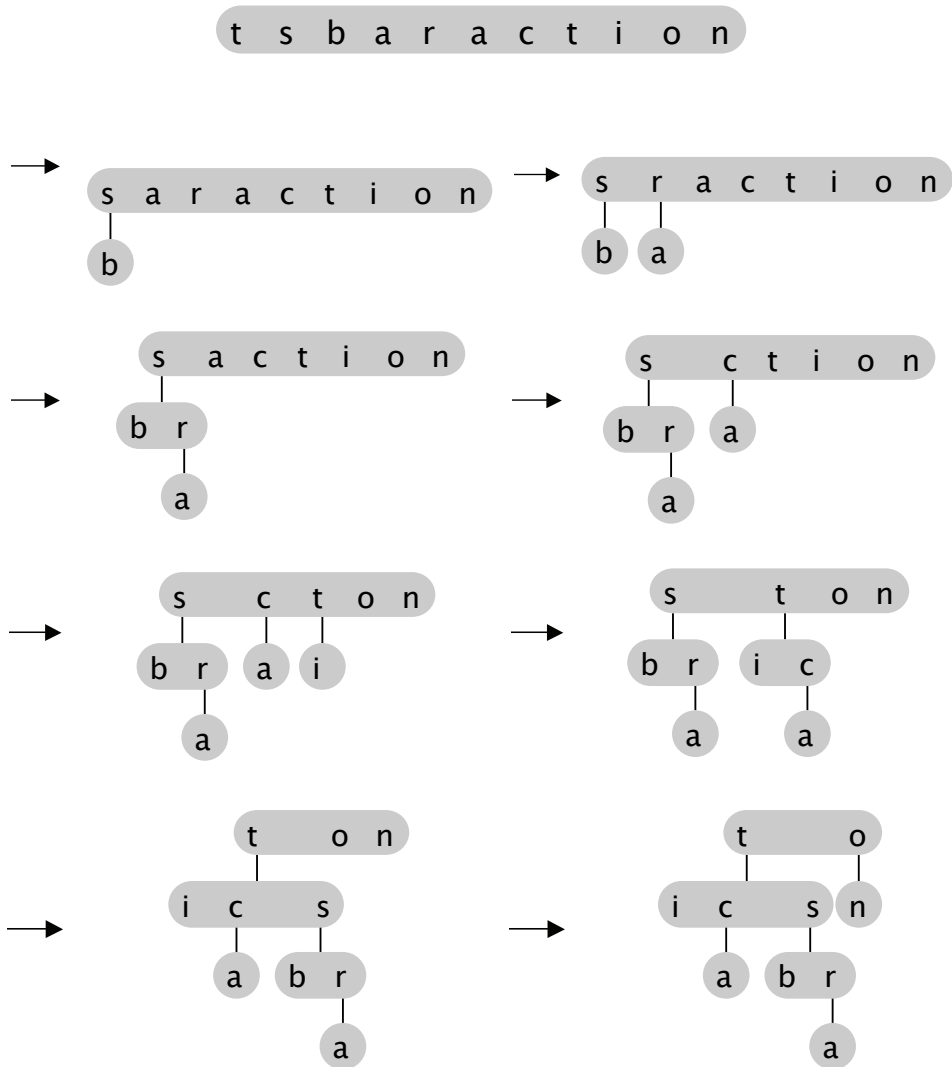
**Figure 28.3**

Illustration of the *consolidate* method following an *extractMax* on a binary heap in which “abstraction” had been inserted.

During this process, each element examined is compared to the root, which is updated to reference the highest priority element, which is guaranteed to be in the root chain at the start of the process. It is easily argued by induction that if the root chain has n elements, then after *consolidate* is called, for any root r with degree d , $|T(r)| = 2^d$. From this it follows that the maximum degree of any root is $\lfloor \log_2 n \rfloor$.

Before presenting the *consolidate* code, we provide a few auxiliary methods. The *moveToRootChain* method takes x , the root of a Fibonacci heap to move to the root chain. It removes $T(x)$ from its current position, and adds it to the root chain. This method requires x has a parent (i.e., $x.parent$ is not *null*).

```
void moveToRootChain(HeapNode<E> x) {
    x.removeFromChain();
    mergeWithRoot(x);
}
```

Correctness Highlights: By the correctness of the node *removeFromChain* method and the inherited *mergeWithRoot*, CHILDREN, DEGREE, PARENT, SIBLINGCHAIN, ROOTCHAIN, and MARKED are preserved. ■

The *link* method takes $r1$, the root of $T(r1)$, and $r2$, the root of $T(r2)$. It makes the smaller of the two roots a child of the other, and returns a reference to the node with the higher priority element among $r1$ and $r2$. It requires that neither $r1$ nor $r2$ are *null*. This method is applied when *root* may not be referencing the highest priority element. It helps restore ROOTCHAIN by ensuring that *root* always references some element in the root chain.

```
FibonacciHeapNode<E> link(FibonacciHeapNode<E> r1, FibonacciHeapNode<E> r2) {
    FibonacciHeapNode<E> parent = r1;
    FibonacciHeapNode<E> child = r2;
    if (comp.compare(r1.element, r2.element) < 0) {
        parent = r2;
        child = r1;
    }
    if (root == child)
        root = parent;
    child.removeFromChain();
    parent.addChild(child);
    return parent;
}
```

Correctness Highlights: Since *parent* is set to the higher priority element, HEAPORDERED is preserved. Also, ROOT is preserved, by resetting the root when *parent* has a higher priority than *root*. By the correctness of the heap node methods *removeFromChain* and *addChild*, CHILDREN, DEGREE, PARENT, SIBLINGCHAIN, ROOTCHAIN, and MARKED are also preserved. ■

During the consolidation process we use the array *bucket* where *bucket[d]* references a root with degree d . To avoid reallocating this array each time, we allocate it once, and then adjust the size as needed. Finally, *stopSentinel* is added to the root list during the consolidation process to mark when all nodes in the root chain have been processed.

```
private final double PHI = (1 + sqrt(5))/2;           //golden ratio
private FibonacciHeapNode[] bucket = null;           //for consolidate
private FibonacciHeapNode<E> stopSentinel = new FibonacciHeapNode<E>(null);
```

The *prepareToConsolidate* method inserts the stop sentinel just left of *root* in the root chain, and adjusts the size of *bucket* as appropriate. If *bucket* has not yet been allocated or it is too small, it is allocated to be one larger than necessary. This will allow *n* to increase by a multiplicative factor of $\phi \approx 1.618$ before it must be resized. If *bucket* is oversized by a factor of two (and larger than 8) its size is reduced to the needed value.

```
final void prepareToConsolidate() {
    stopSentinel.setLeft(root.sibL);           //marks where to stop
    root.setLeft(stopSentinel);
    int degreeUpperBound = (int) ceil(log(getSize())/log(PHI)) + 1;
    if (bucket == null || bucket.length < degreeUpperBound)
        bucket = new FibonacciHeapNode[degreeUpperBound+1];    //okay until n doubles
    else if (bucket.length > 2*degreeUpperBound && bucket.length > 8)
        bucket = new FibonacciHeapNode[degreeUpperBound];
}
```

Correctness Highlights: By ROOTCHAIN and the correctness of *setLeft*, the stop sentinel is placed in the root chain just left of the root. As proven in Section 28.5, when other modifications made to the structure of the Fibonacci heap, it is guaranteed that all nodes have degree in $\{0, 1, \dots, \lceil \log_{\phi} n \rceil\}$. Thus, for any reachable node *x*, *bucket[x.degree]* will exist. ■

Now we present *consolidate* that follows the process described earlier where *ptr* references the node currently being processed. It requires that a maximum priority element is in the root list and that ROOT is the only property that may not hold. Since the linking process may cause the node referenced by *ptr* to become a child of the tree referenced by *bucket[ptr.degree]*, before the linking process begins, a reference to the next node in the root chain is saved.

```
private void consolidate() {
    prepareToConsolidate();           //initialize
    FibonacciHeapNode<E> ptr = (FibonacciHeapNode<E>) root;
    do {
        if (comp.compare(ptr.element, root.element) > 0)           //restore Root
            root = ptr;
        int degree = ptr.degree;
        FibonacciHeapNode<E> next = (FibonacciHeapNode<E>) ptr.sibR;
        while (bucket[degree] != null) {
            ptr = link(ptr, bucket[degree]);           //link existing under current
            bucket[degree] = null;                   //move out of current bucket
            degree++;                                   //degree has now grown by 1
        }
        bucket[degree] = ptr;           //store T(ptr) is bucket degree
        ptr = next;
    } while (ptr != stopSentinel);           //continue until stop sentinel is reached
    stopSentinel.removeFromChain();           //remove the stop sentinel
    Arrays.fill(bucket, null);
}
```

Correctness Highlights: By the correctness of *link* it is easily seen that this method preserves CHILDREN, DEGREE, PARENT, SIBLINGCHAIN, ROOTCHAIN, and MARKED. By

ROOTCHAIN, all nodes in the root chain are considered during the do-while loop. Furthermore, by HEAPORDERED, a maximum priority element must be a node in the root chain. Thus this method restores ROOT, and also ensures that no two trees in the root chain have the same degree. Finally, all buckets are reinitialized to *null* through *Arrays.fill*. ■

28.4.3 Content Mutators

Methods to Perform Insertion

As with a leftist heap and a pairing heap, a Fibonacci heap supports a very fast merge. The *splice* method used within *merge* and also by methods that update the priority of an element. The method takes *a*, a reference to a heap node, and *b*, a reference to a heap node in a different chain. It splices together the chains that contain *a* and *b*, to form a single chain.

```
void splice(HeapNode<E> a, HeapNode<E> b) {
    HeapNode<E> temp = b.sibL;
    b.setLeft(a.sibL);
    a.setLeft(temp);
}
```

Correctness Highlights: Through the use of *setLeft*, SIBLINGCHAIN and ROOTCHAIN (if *a* or *b* are in the root chain) are preserved. ■

The *merge* method takes *rootA*, the root of one Fibonacci heap, and *rootB*, the root of the other Fibonacci heap, and returns a reference to the root of the merged Fibonacci heap. It requires that *T(rootA)* and *T(rootB)* are valid Fibonacci heaps.

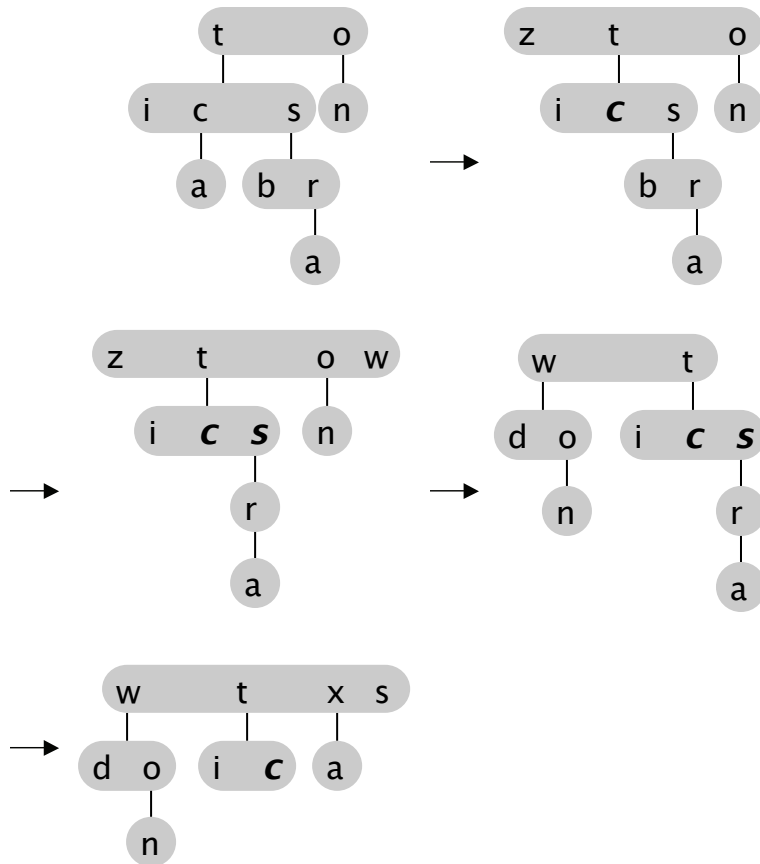
```
HeapNode<E> merge(HeapNode<E> rootA, HeapNode<E> rootB) {
    splice(rootA, rootB);
    if (comp.compare(rootA.element, rootB.element) > 0)
        root = rootA;
    else
        root = rootB;
    return root;
}
```

Correctness Highlights: By the correctness of *splice*, ROOTCHAIN is preserved. Resetting *root* to the higher priority element among the two roots preserves ROOT. No other properties are affected except SIZE which must be preserved by the method that calls *merge*. ■

Methods to Modify Elements

The *add* method is inherited. Recall that it just allocates a new node and merges it with the existing Fibonacci heap. This corresponds to just adding the new node to the root list, and letting *root* refer to the new node if it has a higher priority than the current *root*. We now describe how the Fibonacci heap is modified when an element is updated. A sequence of updates is illustrated in [Figure 28.4](#).

Recall that the *increasePriority* method takes *x*, a reference to the heap node holding element *e* and *element*, the new element to replace *e*. This method requires that *element* $\geq$ *x.element* and that

**Figure 28.4**

Fibonacci heap example of updating the values (using locators). First one of the “a”s is changed to “z.” Next “b” is changed to “w.” Then the “z” is changed to a “d,” and finally one of the “t”s is changed to an “x.”

x is not *null*. If the higher priority for x would violate *HEAPOORDERED*, then $T(x)$ is detached, and merged with the root list. Furthermore *cut* is called to preserve *MARKED*.

```

protected void increasePriority(HeapNode<E> x, E element){
    x.element = element;
    FibonacciHeapNode<E> parent = ((FibonacciHeapNode<E>) x).parent;
    if (parent != null && comp.compare(x.element, parent.element) > 0)
        cut((FibonacciHeapNode<E>) x);           //preserves Marked
    if (comp.compare(element, root.element) > 0)   //preserve Root
        root = x;
}

```

Correctness Highlights: By the correctness of *moveToRootChain*, all properties are preserved except for *MARKED* and *ROOT*. In particular, observe that *HEAPOORDERED* is preserved by moving $T(x)$ to the root chain when x has a higher priority than its parent after it is updated. By the correctness of *cut*, *MARKED* is preserved. Finally, the last conditional preserves *ROOT*. ■

As in a pairing heap, when an element is replaced by a lower priority element, to preserve *HEAPOORDERED*, all if its children are detached and added to the root chain. The method *MoveChildrenToRootChain* takes x , a reference to a heap node, and moves all children of x into the root chain. It returns *true* if and only if x had at least on child, and requires x is not *null*.

```

boolean moveChildrenToRootChain(HeapNode<E> x){
    FibonacciHeapNode<E> ptr = (FibonacciHeapNode<E>) x.child;
    if (ptr == null)                               //by prop Child has no children
        return false;
    do {                                           //preserve Parent
        ptr = (FibonacciHeapNode<E>) ptr.sibR;
        ptr.parent = null;
    } while (ptr != x.child);
    x.child = null;                               //preserve Child
    ((FibonacciHeapNode<E>) x).degree = 0;        //preserve Degree
    splice(root, ptr);                             //preserves RootChain
    return true;
}

```

Correctness Highlights: By *CHILDREN*, if $x.child$ is *null*, then x has no children that need to be moved, and *false* is correctly returned. Otherwise, by *SIBLINGCHAIN*, the loop sets the parent to *null* for all children of x , which preserves *PARENT*. Since x will have no children when this method completes, $x.child$ and $x.degree$ are updated to preserve *CHILD* and *DEGREE*. By the correctness of *splice*, all of the children are spliced into the root chain in a way that preserves *ROOTCHAIN*. ■

When the consolidation process is applied over a set of singleton nodes, it constructs trees such that when the root of a tree has degree d , the tree has 2^d nodes. This observation is easily proved by induction, since a tree of 2^{d+1} nodes is obtained by merging two trees of degree d that have 2^d nodes each. This structure is important since it guarantees that all nodes have degree at most $\log_2 n$. However, when a node is detached from its parent in order to remove it or update its priority, these structural guarantees no longer hold. *MARKED* is designed to limit the extent to which this ideal structure can be modified. Recall that *MARKED* states that for each reachable heap node x not in

the root chain, $x.marked$ is *true* if and only if exactly one child has been removed from x since the last time x 's parent changed (or since it was created). To preserve this property, whenever there is a need to detach a second child from a marked node, the marked node is moved to the root chain. The advantage of moving the node to the root chain, is that the marked node receives a new parent so MARKED is preserved, and also it is easily seen that the other properties, except ROOT, are easily preserved by detaching a subtree and adding it to the root list. The *consolidate* method both restores ROOT and restructures the tree for future efficiency.

Sometimes the parent of a node whose priority is changing is already marked, in which case the parent must be moved to the root chain to preserve MARKED. When a node is moved to the root chain we say it is **cut**. As each node is cut, if its parent is marked then the parent must be cut as well, and this process could propagate up until the parent is already in the root chain. This process of applying recursive cuts is typically called **cascading cuts**. The mark bit, in conjunction with cascading cuts to preserve MARKED, is fundamental to the fast amortized cost for methods such as *increasePriority*.

The *cut* method takes x , a heap node that is to be cut away from its parent. It performs cascading cuts to preserve MARKED. It requires that x is not *null*.

```
private void cut(FibonacciHeapNode<E> x) {
    FibonacciHeapNode<E> parent = x.parent;
    if (parent != null) {                                     //if x not already in root list
        x.removeFromChain();                                 //remove it from where it is
        moveToRootChain(x);                                 //and put int root list
        if (!parent.marked)                                  //if parent is not marked
            parent.marked = true;                           //then mark to denote a child has been cut
        else                                                 //recursively cut parent
            cut(parent);
    }
}
```

Correctness Highlights: If x has no parent, then no computation is required. By the correctness of *removeFromChain* and *moveToRootChain*, x is moved from its current chain to the root chain. If *parent* is not marked, then setting *marked* to *true* preserves MARKED. Otherwise, to preserve MARK the parent must be recursively cut. ■

The method *decreasePriority* takes x , a reference to the Fibonacci heap node holding element e and *element*, the new element to replace e . This method requires that $element \leq x.element$. It requires x is not *null*. If x has no children, then just replacing the element preserves HEAPORDERED, as well as the other properties. Otherwise, the children are moved to the root chain, and then x is moved to the root chain. The performance guarantees require that *consolidate* be applied after moving the children. Also when the priority for the root is decreased, *consolidate* is applied to both find a new highest priority element, restructuring the tree in the process for future efficiency.

```
void decreasePriority(HeapNode<E> x, E element) {
    FibonacciHeapNode<E> node = (FibonacciHeapNode<E>) x;    //reduce casting
    node.element = element;                                    //replace element
    if (moveChildrenToRootChain(node)) {                       //Case 1: move children to root chain
        cut(node);                                             //apply cascading cut to preserve Marked
        consolidate();                                         //consolidate root chain, and preserve Root
    } else if (node == root)                                   //Case 2
        consolidate();                                         //need to find max priority element in root list
}
```

Correctness Highlights: By the correctness of *moveChildrenToRootChain*, *cut*, and *consolidate*, if x has children then this method or if the *root* is being removed, CHILDREN, DEGREE, PARENT, SIBLINGCHAIN, ROOTCHAIN, and MARKED are preserved. By PARENT, if the neither of the two cases occur then x is in the root chain. Thus, by ROOT and HEAPORDERED, it cannot have a higher priority than *root*, so all properties hold without any computation needed beyond replacing the element. ■

Methods to Perform Deletion

We now describe the method to remove elements. Like the method for increasing the priority of an element, *remove* uses cascading cuts in conjunction with the consolidate method. The internal *remove* method takes x , a reference to the heap node to remove, and removes x from the Fibonacci heap. It returns the element held in x . This method requires that x is not *null*.

```
E remove(HeapNode<E> x) {
    if (getSize() == 1) {                                //special case when singleton element is being removed
        root = null;
    } else {
        moveChildrenToRootChain(x);                      //preserve HeapOrdered
        if (root == x)                                   //make sure root is somewhere in root chain
            root = root.sibR;
        cut((FibonacciHeapNode<E>) x);                  //move x to root chain
        x.removeFromChain();                             //remove x
        consolidate();
    }
    x.markDeleted();                                    //preserve Removed
    x.prev.setNext(x.next);                             //preserve IterationList
    size--;                                              //preserve Size
    return x.element;
}
```

Correctness Highlights: By the correctness of *moveChildrenToRootChain*, *cut*, *consolidate*, and the node *removeFromChain* methods, HEAPORDERED CHILDREN, DEGREE, PARENT, SIBLINGCHAIN, ROOTCHAIN, and MARKED are preserved. In particular, for *consolidate* to restore ROOT it requires that HEAPORDERED holds and that *root* references some element in the root chain – both of those hold when *consolidate* is called. By the correctness of the *markDeleted* method, REMOVED is preserved. It is also easily checked that ITERATIONLIST and SIZE are preserved. ■

28.5 Performance Analysis

The asymptotic time complexities of all public methods for the FibonacciHeap class are shown in Table 28.5, and the asymptotic time complexities for all of the public methods of locator are given

method	amortized time complexity	worst-case time complexity
<i>ensureCapacity(x)</i> <i>max()</i>	$O(1)$ $O(1)$	$O(1)$ $O(1)$
<i>add(o)</i> <i>addTracked(o)</i>	$O(1)$ $O(1)$	$O(1)$ $O(1)$
<i>extractMax()</i>	$O(\log n)$	$O(n)$
<i>clear()</i> <i>contains(o)</i> <i>getEquivalentElement(o)</i> <i>getLocator(o)</i> <i>remove(o)</i> <i>toArray()</i> <i>toString()</i> <i>trimToSize()</i> <i>accept()</i>	$O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$	$O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$ $O(n)$
<i>addAll(c)</i>	$O(c \log n)$	$O(n)$
<i>retainAll(c)</i>	$O(c n)$	$O(c n)$

Table 28.5 Summary of the asymptotic time complexities for the *PriorityQueue* public methods when implemented using a Fibonacci heap.

in Table 28.6.

Observe that the Fibonacci heap allocates one heap node for every element. For a threaded Fibonacci heap, as we have implemented here, there are nine instance variables per node. So the space usage is roughly that of $9n$ references. (In an unthreaded implementation there would be only seven instance variables per node).

Following the basic approach presented in Cormen et al. [42], we use the **potential method** to compute the amortized cost. Appendix B reviews amortized complexity and methods of doing the analysis including the potential method. For a Fibonacci Heap H , we use the potential function $\Phi(H) = t(H) + 2m(H)$ where $t(H)$ is the number of trees in the root chain of H , and $m(H)$ is the number of marked nodes in H . We use $D(n)$ to denote the maximum degree (number of children) of a node in an n node Fibonacci heap. When the only mutators supported are *insert* and *extractMax*, it can be proven using induction that $D(n) \leq \lfloor \log_2 n \rfloor$. The crux of the argument is to show that for any node x , $|T(x)| \geq 2^{x.degree}$. At the end of this section, we show that even when all methods are supported, $D(n) = O(\log n)$. We first analyze the worst-case cost and amortized cost for all of the public methods given this bound on $D(n)$.

Throughout our discussion we let H denote the Fibonacci heap before the method executes, and let H' denote the Fibonacci heap after the method executes. Recall that the amortized cost is the worst case over H of

$$\Phi(H') - \Phi(H) + f(H)$$

where $f(H)$ is the actual, worst-case cost for executing the given method on H .

We first consider the amortized cost of adding a node. Since the number of marked nodes does not change, and the root chain increases by one, $\Phi(H') - \Phi(H) = 1$. Since the actual cost is constant, it follows that the amortized cost is also constant.

Next we consider the cost of taking the union of H_1 and H_2 to get H . Since the root chains are just spliced together, the number of trees in $t(H) = t(H_1) + t(H_2)$. Since the state of which nodes are marked is not changed, it also follows that $m(H) = m(H_1) + m(H_2)$, so $\Phi(H) = (\Phi(H_1) +$

method	amortized time complexity	worst-case time complexity
<i>get()</i>	$O(1)$	$O(1)$
<i>advance()</i>	$O(1)$	$O(n)$
<i>hasNext()</i>	$O(1)$	$O(n)$
<i>next()</i>	$O(1)$	$O(n)$
<i>retreat()</i>	$O(1)$	$O(n)$
<i>update(o), increase priority</i>	$O(1)$	$O(n)$
<i>remove()</i>	$O(\log n)$	$O(n)$
<i>update(o), decrease priority</i>	$O(\log n)$	$O(n)$

Table 28.6 Summary of the time complexities for the locator methods of the Fibonacci heap that use the pairing heap Tracker inner class.

$\Phi(H_2)) = 0$. Finally, the actual cost is constant since splicing the root chains takes constant time, comparing the priorities of the roots from H_1 and H_2 takes constant time, and so does the rest of the computation. Thus, merging two Fibonacci heaps has amortized and worst-case constant cost.

We now argue that the amortized cost of *extractMax* is $O(D(n)) = O(\log n)$. In the worst case, there could be $D(n)$ children processed by *consolidate* by the outer loop. We now consider the inner loop for the consolidation process. The size of the root chain when *consolidate* begins is at most $D(n) + t(H) - 1$ due to the original $t(H)$ roots minus the extracted one, plus the children of the extracted one. Each iteration of the inner loop links one root to another which takes constant time. So the worst-case cost is $O(D(n) + t(H)) = O(\log n)$. We now analyze the amortized cost. Since there is no change to the marked nodes $\Phi(H') - \Phi(H) = t(H') - t(H)$. Observe that $t(H') \leq D(n) + 1$ since there is at most one node of each degree, and each degree is in $\{0, 1, \dots, D(n)\}$. Thus $\Phi(H') - \Phi(H) \leq D(n)$ since there $t(H) \geq 1$. Thus, the amortized cost is $O(\log n)$ since both the change of potential and the actual costs are logarithmic.

Next we analyze *update* when called for a node x for which the priority increases. The actual cost is dominated by the cost of performing the cascading cuts. Suppose there are k calls made to *cut*. Then the actual cost is $O(k)$. We now compute the change in potential. We argue that $t(H') = t(H) + k$. Along with the original $t(H)$ trees, there is a new tree rooted at x , plus $k - 1$ trees produced by the $k - 1$ recursive calls made by the cascading cuts. Also note that $m(H') \leq m(H) - k + 2$ since either $k - 1$ or k nodes are unmarked by the cascading cuts (depending on whether it ends with marking an unmarked node or reaching the root chain). Thus,

$$\begin{aligned} t(H') - t(H) &\leq t(H) + k + 2(m(H) - k + 2) - (t(H) + 2m(H)) \\ &= t(H) + k + 2m(H) - 2k + 4 - t(H) + 2m(H) \\ &= 4 - k. \end{aligned}$$

So the amortized cost is constant. The worst case cost is the height of the tree, which in the worst case could be linear.

The analysis for removing node x combines aspects of that for *extractMin* and *update* when the priority of the element is increased. As with *extractMin*, the time to remove all children and move them to the root list is proportional to the number of children of x . Although the sibling chain for x 's children can be placed into the root list in constant time, each child of x must have its parent set to *null*. This portion of remove increases the potential by one for each child of x , which is $O(\log n)$. The other aspect of remove is that when x is detached from its parent, up to k calls to *cut* might occur. However, exactly as shown above, the amortized cost for these cuts is constant. Thus the overall amortized cost for removing a node once a reference to it is obtained is $O(\log n)$. Because

of the possibility of the cascading cuts propagating up the tree, the worst-case cost actual cost is linear.

The analysis for *update* when the priority of an element is decreased is exactly like *remove*, except the upper bound for $t(H')$ is bigger by one since the node being updated is moved to the root chain along with its children.

What remains is to upper bound $D(n)$. In particular, we show that $D(n) \leq \lceil \log_\phi n \rceil$ where $\phi = (1 + \sqrt{5})/2$. We start with the following simple claim.

Claim: Let x be a heap node for which $x.degree = d$. We let $y_1, \dots, y_k$ denote the children of x , in the order they were linked to x from earliest to latest. Then $y_1.degree \geq 0$, and for $i = 2, \dots, d, y_i.degree \geq i - 2$.

Proof: We prove this by induction on i . Clearly $y_1.degree \geq 0$. For $i \geq 2$, when y_i is linked to x , all of $y_1, \dots, y_{i-1}$ were children of x , and so $y_i.degree = i - 1$. Also two nodes are only linked if they have the same degree at the time when linked. Since a node can lose at most one child, before it is cut itself, it follows that $y_i.degree \geq i - 2$. ■

The **Fibonacci sequence** is defined as follows, where we use F_k to denote the k^{th} Fibonacci number.

$$F_k = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ F_{k-1} + F_{k-2} & \text{otherwise.} \end{cases}$$

Using two independent inductive proofs, it is easily shown that $F_{k+2} = 1 + \sum_{i=0}^k F_i \geq \phi^k$ where ϕ is the **golden ratio** which is $(1 + \sqrt{5})/2 \approx 1.618$.

The name Fibonacci heap is derived from the use of the Fibonacci sequence in the analysis. We now prove that for any heap node $D(n) = O(\log n)$. To prove this bound on $D(n)$, we prove the following lemma.

Lemma: For x any node in a Fibonacci heap, and for $d = x.degree$, $|T(x)| \geq F_{d+2} \geq \phi^d$ for $\phi = (1 + \sqrt{5})/2$.

Proof: Let s_k denote the minimum possible value for $|T(z)|$ over all nodes z for which $z.degree = d$. As we did earlier, let $y_1, y_2, \dots, y_d$ denote the children of x in the order in which they were linked to x . We want to compute a lower bound for $|T(x)|$.

$$|T(x)| \geq s_d = 2 + \sum_{i=2}^d s_{y_i.degree} \geq 2 + \sum_{i=2}^d s_{i-2}$$

where the last line follows from the earlier claim, and the fact that clearly s_k increases monotonically with k .

Using induction on d , we show that $s_d \geq F_{d+2}$ for all $d \geq 0$. The base cases for $d = 0$ and $d = 1$ are easily shown. For the inductive step, we assume $d \geq 2$ and that $s_i \geq F_{i+2}$ for $i = 0, \dots, d - 1$. We have

$$s_d \geq 2 + \sum_{i=2}^d s_{i-2} \geq 2 + \sum_{i=2}^d F_i = 1 + \sum_{i=0}^d F_i = F_{d+2}.$$

This completes the proof, since we have shown that $|T(x)| \geq s_d \geq F_{d+2} \geq \phi^d$. ■

As a corollary to this lemma we can prove that $D(n) = O(\log n)$. Let x be any heap node, and let $x.degree = d$. We have $n \geq |T(x)| \geq \phi^d$. Taking base- ϕ logarithms of both sides yields that $d \leq \log_\phi n$. (Since d is an integer, the stronger claim that $d \leq \lceil \log_\phi n \rceil$ holds.) Thus the maximum degree for any node is at most $\log_\phi n = O(\log n)$.

28.6 Quick Method Reference

FibonacciHeap Public Methods

- p. 428 *FibonacciHeap*()
- p. 428 *FibonacciHeap*(Comparator<? super E> comp)
- p. 98 void **accept**(Visitor<? super E> v)
- p. 410 void **add**(E element)
- p. 100 void **addAll**(Collection<? extends E> c)
- p. 410 PriorityQueueLocator<E> **addTracked**(E element)
- p. 100 void **clear**()
- p. 97 boolean **contains**(E value)
- p. 99 void **ensureCapacity**(int capacity)
- p. 414 E **extractMax**()
- p. 96 int **getCapacity**()
- p. 96 Comparator<? super E> **getComparator**()
- p. 98 E **getEquivalentElement** (E target)
- p. 414 PriorityQueueLocator<E> **getLocator**(E element)
- p. 96 int **getSize**()
- p. 96 boolean **isEmpty**()
- p. 414 PriorityQueueLocator<E> **iterator**()
- p. 406 E **max**()
- p. 414 boolean **remove**(E element)
- p. 100 void **retainAll**(Collection<E> c)
- p. 97 Object[] **toArray**()
- p. 97 E[] **toArray**(E[] array)
- p. 98 String **toString**()
- p. 99 void **trimToSize**()

FibonacciHeap Internal Methods

- p. 97 int **compare**(E e1, E e2)
- p. 432 void **consolidate**()
- p. 436 void **cut**(FibonacciHeapNode<E> x)
- p. 411 void **decreasePriority**(HeapNode<E> x, E element)
- p. 97 boolean **equivalent**(E e1, E e2)
- p. 407 HeapNode<E> **find**(E element)
- p. 411 void **increasePriority**(HeapNode<E> x, E element)
- p. 408 HeapNode<E> **insert**(E element)
- p. 431 FibonacciHeapNode<E> **link**(FibonacciHeapNode<E> r1, FibonacciHeapNode<E> r2)
- p. 408 HeapNode<E> **merge**(HeapNode<E> rootA, HeapNode<E> rootB)
- p. 411 HeapNode<E> **mergeQueue**()
- p. 408 void **mergeWithRoot**(HeapNode<E> x)
- p. 410 Queue<HeapNode<E>> **moveChildrenToQueue**(HeapNode<E> x)
- p. 435 boolean **moveChildrenToRootChain**(HeapNode<E> x)
- p. 431 void **moveToRootChain**(HeapNode<E> x)
- p. 406 HeapNode<E> **newHeapNode**(E value)
- p. 432 void **prepareToConsolidate**()
- p. 406 void **reinitialize**()
- p. 413 E **remove**(HeapNode<E> x)
- p. 433 void **splice**(HeapNode<E> a, HeapNode<E> b)
- p. 99 void **traverseForVisitor**(Visitor<? super E> v)
- p. 413 void **update**(HeapNode<E> x, E element)
- p. 98 void **writeElements**(StringBuilder s)

Chapter 29

Ordered Collection ADT

package collection.ordered

Collection<*E*>

↳ **OrderedCollection**<*E*>

An ordered collection is an untagged algorithmically positioned collection of comparable elements that may contain duplicates. That is, order is defined by a comparator and the collection may contain multiple “equal” elements and multiple occurrences of the same element. The primary methods are to add an element, determine if an element is in the collection, to remove an element, and to find the previous or next element in the ordering defined by the comparator. The iteration order must follow the ordering defined by the comparator. Data structures for an ordered collection typically provide logarithmic time implementations for these methods. However, in some cases the data structures support constant time implementations, or require linear time implementations.

An ordered collection supports methods that concern order such as *min*, *max*, *predecessor*, and *successor* and methods that allow the user application to efficiently perform queries such as a **range search** in which one can iterate through all elements in the collection between a specified range of values.

Most OrderedCollection ADT data structures use a rooted tree as their internal representation. However, list-based and array-based implementations are also possible. The skip list data structure augments a sorted list to support logarithmic time navigation. An array-based ordered collection provides the most space efficient solution, although it incurs worst-case linear time cost for inserting and removing elements. We present several tree-based implementations, as well as the skip list and array-based alternatives.

29.1 Case Study: Historical Event Collection (Range Queries)

In Section 1.5, we discussed the conceptual design for a simple historical event collection. Recall that the task is to maintain a collection of historical event objects where each event object includes a date, a description, and possibly other information about the event. The earlier discussion, which focused on the conceptual design, illustrated how evolving application requirements can affect the selection of an appropriate ADT.

In this section, we discuss the ADT and data structure decision process to support the following operations:

1. Insert a new historical event into the collection.
2. Given a date, return a list of all events in the collection that occurred on that date.
3. Given a date, remove all events with that date.
4. Given a date and description, remove all events with the given date and description.

5. Given a date d and a range r , return a list of all events in the collection that occurred in the interval $[d - r, d + r]$.

To support these operations, the historical events must be organized according to the date, so that all events with an equivalent date can be accessed. For these methods there is no need to compare the dates. However, to support the range search (the fifth operation listed), an `OrderedCollection` ADT, in which the comparator uses only the date is the most appropriate data structure. A `Set` ADT and a `PriorityQueue` ADT do not support a range search. One might consider a `DigitizedOrderedCollection` ADT. However, the date is only four digits, and most likely all of them must be examined to answer the query. Furthermore, since the date can be stored as an integer, a single comparison will suffice. Thus there is no benefit to using a digitized ordered collection for this application. Finally, since there is just one feature defining the ordering, the `SpatialCollection` ADT is also not appropriate.

Now that the `OrderedCollection` ADT has been selected to organize the events according to the date, a data structure should be selected. The skip list data structure supports a range search most naturally since within the skip list is a linked list of all events sorted by date. However, if reducing the search time is fundamental, a red-black tree may be more efficient, especially if the number of elements in the specified range is small.

Section 50.1 explores the ADT and data structure options that support the ability to retrieve a list of all events that include a given word in the event description. As part of this design, an ordered collection is used to store all events that contain a given word so that the iteration order for the set of events containing a given word is chronological.

29.2 Case Study: Linux Virtual Memory Map

Virtual memory is an abstraction that provides each process with the illusion that it has exclusive use of the main memory. Each process has a uniform view of memory, which is known as its **virtual address space**. Each virtual address space is composed of a set of regions called **segments**, where each segment is a contiguous portion of allocated virtual memory whose pages are often related in some way. For example, the shared libraries, the data, the code, and the heap are all distinct segments. An important aspect about the layout of the segments is that there can be gaps in the virtual address space. By only maintaining information about virtual pages that have been allocated, the cost within the kernel to maintain the virtual pages depends only upon the number of segments that are allocated. The text of Bryant and O'Hallaron [35] is an excellent source for further information on the Linux virtual memory system.

The kernel data structure for managing the virtual address space includes:

- a **page directory table** which provides a mapping from a virtual address to a **physical address** (a page in physical memory or on disk), and
- a **memory map** that consists of a list of **area structs**, each of which contains information about a segment of the current virtual address space. Included in each area struct is the starting virtual address (*vm.start*) and ending virtual address (*vm.end*) for the segment, as well as other information such as the read and write permission for the segment.

A **page fault** occurs when the executing program references a virtual memory address for a page that is not currently in memory. In this case study we focus on page fault handling. Suppose that a page fault occurs when translating virtual address x to a physical address. This fault causes control to transfer to the kernel's **page fault handler**. The page fault handler must first determine if x is a

legal virtual address. That is, it must determine if there is some allocated segment of virtual memory for which $vm\_start \leq x \leq vm\_end$.

Since a process can have an arbitrary number of virtual memory segments, a linear search for the appropriate segment would be prohibitively expensive. Suppose that the process had p virtual memory segments, where a_{s_i} is vm_start for the i^{th} segment and a_{e_i} is vm_end for the i^{th} segment. Since the virtual memory segments are non-overlapping, to determine if x is a legal virtual address, it is sufficient to store the page directory table as an ordered collection over the segments where the comparator is defined according to vm_start . Observe that x is a legal address if and only if $a_{s_i} \leq x \leq a_{e_i}$ for the largest value of i for which $a_{s_i} \leq x$. Finding this value of i corresponds to finding x if it is in the ordered collection or otherwise finding the predecessor of x . Thus, in $O(\log p)$ time, the kernel can determine if x is a legal virtual address.

Finally, we briefly consider the process of selecting an ordered collection data structure for the memory map. One might imagine using a binary search tree implementation of the ordered collection. However, most applications exhibit a high degree of locality of reference in which virtual memory pages in the same segment would be accessed sequentially, so significant savings can be achieved by selecting a data structure that optimizes access for recently-accessed elements. FreeBSD 6.0, as one example, uses a splay tree. An important advantage of a splay tree for this application is that virtual pages that were recently accessed will be near the root of the tree enabling it to take advantage of virtual page access patterns. Linux [28] uses the red-black tree, which does not favor recently accessed pages but has the advantage of a guaranteed logarithmic upper bound time complexity for servicing each page fault. In contrast, the splay tree only provides an amortized logarithmic cost. In the worst-case it might take linear time to determine if x is a valid virtual address.

29.3 Interface

Our *OrderedCollection* interface is similar to Java's *SortedSet** interface, but it has different semantics for handling duplicate elements. The following methods are added to those inherited from the *Collection* interface (Chapter 7).

OrderedCollection(): Creates a new empty ordered collection using the default comparator.

OrderedCollection(Comparator comp): Creates a new empty ordered collection with the given comparator.

E get(int r): Returns the r^{th} element in the sorted order, where $r = 0$ is the minimum element. It throws an *IllegalArgumentException* when $r < 0$ or $r \geq n$.

E getEquivalentElement(E target): Returns an element in the collection that is equivalent to *target* according to the comparator associated with this collection. It throws a *NoSuchElementException* when there is no equivalent element.

Locator<E> iteratorAtEnd(): Returns a locator that has been initialized to AFT.

E min(): Returns a least element in the collection (according to the comparator). More specifically, the first element in the iteration order is returned. This method throws a *NoSuch-*

*We have omitted some methods provided in Java's *SortedSet* that can be easily implemented using the methods provided here. This enables us to focus on the methods that illustrate how the data structures work.

ElementException when the collection is empty. This method has the same semantics as Java's *SortedSet first* method.

E max(): Returns a greatest element in the collection (according to the comparator). More specifically, the last element in the iteration order is returned. This method throws a *NoSuchElementException* when the collection is empty. This method has the same semantics as Java's *SortedSet last* method.

E predecessor(E x): Returns a greatest element in the ordered collection that is less than x . If x is in the collection, the element before the first occurrence of x in the iteration order is returned. Note that x need not be an element of the collection for *predecessor* to return a value. It throws a *NoSuchElementException* when there is no element in the collection smaller than x . (The element returned would be the last element in the collection returned by the method call *headSet(x)* in Java's *SortedSet* interface.)

E successor(E x): Returns a least element in the ordered collection that is greater than x . If x is in the collection, the element after the last occurrence of x in the iteration order is returned. Note that x need not be an element of the collection for *successor* to return a value. It throws a *NoSuchElementException* when there is no element in the collection larger than x . (The element returned would be the first element in the collection returned by the method call *tailSet(x)* in Java's *SortedSet* interface. If an application needs to iterate through *tailSet(x)*, the *getLocator* method in conjunction with *advance* could be used.

Critical Mutators for OrderedCollection: *add, addTracked, addAll, remove, clear, retainAll*

29.4 Terminology

In addition to the definitions given in Chapter 7, we use the following definitions throughout our discussion of the OrderedCollection ADT and the data structures that implement it.

- When element $e1$ is less than element $e2$ with respect to the comparator associated with this ordered collection, we write $e1 < e2$. Likewise, for the $\leq$, $>$, $\geq$, and $=$ operator between elements $e1$ and $e2$.
- The **rank** r element is the r^{th} element in the sorted order, where $r = 0$ is the minimum element.

Note that maintaining proper order within an ordered collection data structure depends upon the requirement that each object's ordering relative to other objects does not change, unless that change is explicitly managed by a method of the ordered collection. For example, an ordered collection could not maintain an alphabetized ordering of People objects if the "name" field of objects in the collection could be changed (for example, by a *setName* method) without the ordered collection itself becoming aware of the change. Objects in ordered collections need not be immutable. However, it is recommended that fields used for comparison purposes be declared final.

29.5 Competing ADTs

We briefly discuss other ADTs that may be appropriate in a situation when an `OrderedCollection` ADT is being considered.

DigitizedOrderedCollection ADT: A digitized ordered collection is more appropriate than an ordered collection when (1) the elements can be viewed as a sequence of digits, and (2) performing $O(\log n)$ comparisons between elements is more expensive than extracting the average number of distinguishing digits in an element. A digitized ordered collection is also indicated when the application requires either a method to efficiently find all elements in the collection that begin with a given prefix, or a method to find the set of elements in the collection that have a longest prefix in common with a given element.

PriorityQueue ADT: If the ordering defined by the comparator corresponds to a priority and the application primarily accesses the highest priority item, and possibly accesses or removes items using a locator, then a priority queue should be considered. Because a priority queue does not need to support a general purpose search, data structures that implement a priority queue can perform the other methods more efficiently (though asymptotically equivalent) and using less space. However, an ordered collection should be chosen in favor of a priority queue when the required iteration order is based on the ordering of the elements.

Set ADT: If the elements held in the collection are comparable, but the application only needs to locate elements, add elements, and remove elements, then a set is a better choice since it can perform all of these operations in expected constant time. One would use an ordered collection instead of a set when application requires methods that concern an ordering among the elements. In addition, in a set the elements are unique, so an ordered collection or a bucket mapping is preferable when duplicates are required.

TaggedOrderedCollection ADT: If it is more natural for the application to define an ordering based on a tag associated with the element (versus by defining a comparator for the elements held in the collection) and it is uncommon for multiple elements to have the same tag, then an tagged ordered collection is more appropriate.

TaggedBucketOrderedCollection ADT: If it is more natural for the application to define an ordering based on a tag associated with the element (versus by defining a comparator for the elements held in the collection) and it is common for multiple elements to have the same tag, then a tagged bucket ordered collection is more appropriate.

29.6 Selecting a Data Structure

Having settled on the `OrderedCollection` ADT, the next design decision is whether to use a sorted array, some form of a search tree, or a skip list. To select the best ordered collection data structure, consider which of the following properties are most important for the desired application.

Ability to self-adjust to minimize the search time for the particular access pattern. All the ordered collection data structures we present are designed for efficient search time. However, the structure created by most data structures is independent of the **access pattern**, the frequency with which a search is performed for each element in the collection. A splay tree

Memory Type	approximate access time (ns)	cost per megabyte (\$)
cache	5-20	10-75
main memory	60-120	0.50-5.00
secondary storage	20,000,000	.001-0.10

Table 29.1 Approximate access times in nanoseconds (ns) as compared with the cost for cache, main memory, and secondary storage (disk).

modifies the tree structure according to the access pattern to minimize the expected search time for frequently accessed elements under the assumption that the access pattern does not change significantly over time.

Frequency of adding or removing elements from the collection. To minimize search time, balanced search trees (red-black tree, B-tree, B+-tree, AVL tree) perform reorganization when elements are added or removed from the collection. This reorganization helps reduce search time. Thus, balanced search trees are extremely beneficial when the collection remains fairly stable, meaning that many searches take place without intervening reorganization. However, if there is very frequent change to the membership of the collection, then this reorganization takes time, so a data structure that does not perform reorganization, such as a skip list, should be considered.

Frequency of access based on the ordering of the elements. Some ordered collection data structures (sorted array or skip list) can perform *min*, *max*, move forward (*advance*), and move backward (*retreat*) in constant time, whereas data structures that use a search tree require logarithmic time.

Acceptability of having only a good amortized or expected time bound. The splay tree data structure occasionally performs linear-time operations in which significant reorganization of the tree structure is performed. While the insert and search methods have amortized logarithmic cost, in a real-time application occasional linear time reorganization may not be acceptable. The skip list data structure has expected logarithmic cost with extremely low probability of linear time cost. However, because the structure is randomized, no specific sequence of operations can predictably result in poor performance.

Locality. Memory is organized in three levels: **cache**, **main memory**, and **secondary storage**. Main memory and secondary storage are organized in chunks called **pages** which are typically around 2000-4000 bytes. Caches are organized into finer grain chunks called **cache lines** of up to 512 bytes. Table 29.1 shows the trade-offs between these three types of memory. Cache has the fastest access time but is expensive. On the other end of the spectrum, secondary storage has relatively slow access, but is very inexpensive. Whenever a memory location is accessed and it is not in one of the cache lines, it is called a **cache miss**. Operating systems employ a strategy to determine which cache line to remove/flush from the cache when there is a cache miss. Similarly, when an object is accessed that is in the virtual memory but not in main memory, this is called a **virtual memory page fault**. For very large data structures which must reside on disk (in virtual memory or in files), the dominant cost of accessing the desired element is the number of disk pages that must be brought into memory. For such applications, it is important to organize the data structure in a way that minimizes the number of disk pages that must be read from secondary storage.

Since the top levels of search trees are accessed with each method, they will remain in cache, so search trees have good locality properties. Sorted arrays and skip lists will tend to have more cache misses and page faults, which increases their search cost.

Since splay trees move frequently accessed data near the root of the tree, frequently accessed items will remain in cache. Thus, a splay tree is a particularly good choice when the data structure is large but accessing an element once increases the likelihood that it will be accessed again in the near future.

Size of the collection. When the number of elements held in the collection gets so large that even a data structure that just maintains the references to the elements cannot fit in main memory, the data structure must reside in secondary storage. Some data structures (B-tree, B+-tree) are designed specifically to reduce the number of disk pages that must be accessed for such collections.

29.7 Summary of Ordered Collection Data Structures

This book provides the following ordered collection implementations. This section summarizes the strengths and weaknesses of each of these data structures in terms of application requirements. [Table 29.2](#) provides a visual summary of these trade-offs.

SortedArray: The sorted array provides very efficient use of space and the fastest search time independent of the access pattern. Specifically, a binary search (Section 30.3.3) examines at most $\lfloor \log_2(n + 1) \rfloor$ elements to locate a desired element. Also, the sorted array provides constant time access to the element at a given rank. However, this data structure has a very significant drawback: adding or removing an element requires shifting all elements that follow it. Thus, this data structure is a good choice only when the elements added or removed are near the maximum, or when there are few mutations after inserting all elements.

BinarySearchTree: When elements are inserted into a binary search tree in a random order, then it has the desirable property of having logarithmic height, where the **height** is the maximum length of a path from the root to a leaf. The height is important since the cost of most methods is proportional to the height of the tree. In the best case, a binary search tree has height $\lfloor \log_2(n + 1) \rfloor$. If the elements are inserted in a random order then the expected height is roughly $2 \ln n \approx 1.386 \log_2 n$ [97]. However, the binary search tree can degenerate to a sorted list (which has a height of n). To avoid this situation, most implementations that build upon a binary search tree (red-black tree, B-Tree, B+-Tree) are **balanced**, meaning that they include methods that reorganize the tree so that the worst-case height is logarithmic.

RedBlackTree: The red-black tree is a balanced binary search tree in which a single bit (a color of red or black) associated with each tree node is used to ensure that the number of comparisons made when searching for any element is at most $2 \log_2 n$. Through partial analysis and simulations, it has been shown that a search in a red-black tree constructed from n elements, inserted in a random order, uses about $1.002 \log_2 n$ comparisons, on average [136]. The red-black tree is the best general-purpose ordered collection data structure since it has guaranteed worst-case logarithmic bounds on the tree height and has very fast search time. If it is important to have constant time methods to reach the previous or next elements in the collection, additional structure can be added for this purpose at the expense of increased execution time and space usage.

SplayTree: A splay tree is a form of a balanced binary search tree in which the nodes store no explicit information to enforce a balancing condition. Instead, a sequence of **rotations** is used to restructure the search tree so that the element accessed in the last method call is brought to

Key								
⦿ Excellent								
○ Very Good								
● Fair								
• Method does nothing								
Method		SortedArray	BinarySearchTree	RedBlackTree	SplayTree	Btree	B+Tree	Skiplist
Ordered Collection Methods	add(o)	●	●	○	○	○	○	○
	addAll(c), per element	○	●	○	○	○	○	○
	clear(), per element	⦿	⦿	⦿	⦿	⦿	⦿	⦿
	contains(o)	○	●	○	○	○	○	○
	ensureCapacity(x)	●	•	•	•	•	•	•
	getLocator(o)	○	●	○	○	○	○	○
	iterator(), iteratorAtEnd()	⦿	○	○	○	○	⦿	⦿
	max(),min()	⦿	○	○	○	○	⦿	⦿
	prededessor(o)	⦿	○	○	○	○	⦿	⦿
	remove(o), retainAll(c), per element	●	●	○	○	○	○	○
	successor(o)	⦿	○	○	○	○	⦿	⦿
Other Issues	trimToSize()	●	•	•	•	•	•	•
	accept(v), toArray(), toString(), per element	⦿	⦿	⦿	⦿	⦿	⦿	⦿
	typical space	⦿	○	○	○	○	●	○
	randomized							✓
	amortized approach (occasionally restructures)	✓						
Locator Methods	fast access to recently accessed elements	✓						
	designed to minimize disk pages read						✓	✓
	advance()	⦿	○	○	○	○	⦿	⦿
	get()	⦿	⦿	⦿	⦿	⦿	⦿	⦿
	hasNext(), next()	⦿	○	○	○	○	⦿	⦿
	remove()	●	●	○	○	○	○	⦿
	retreat()	⦿	○	○	○	○	⦿	⦿
Time Complexity								
⦿ O(1) time								
○ O(log n) time								
● O(n) time								
• method does nothing								
Space Usage								
⦿ Excellent								
○ Very Good								
● Good								

Table 29.2 Trade-offs among the data structures that implement the OrderedCollection ADT.

the root of the tree. It can be shown that if a random sequence of elements is accessed, then this restructuring leads to trees with expected logarithmic height. Thus, recently accessed elements are near the root and can be located very efficiently. The amortized complexity for all splay tree methods is logarithmic. However, as with a binary search tree, a splay tree can degenerate into a sorted list. The splay tree should be used if the goal is to minimize the expected access time, provided that the access pattern is stable and not uniform (i.e., some elements are accessed much more frequently than others), and provided that elements recently accessed are more likely to be accessed again soon.

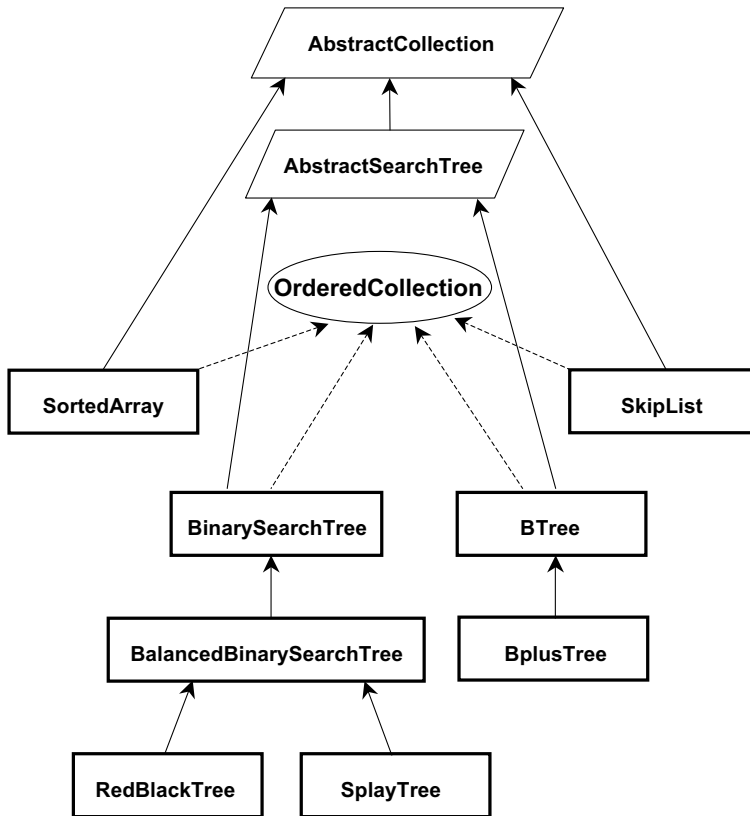
B-Tree: A B-tree is an extension of a balanced binary search tree in which each node can hold between $t - 1$ and $2t - 1$ elements, where integer $t > 1$ is provided as a parameter to the constructor. In a binary search tree, each node holds a single element. The B-tree is designed for collections that are so large that secondary storage must be used. The larger node size (chosen with knowledge of the page size) helps minimize the number of disk pages that must be read to locate an element.

B+-Tree: The B+-tree is variation of a B-tree in which the internal nodes are used only for navigation. The leaves (from left to right) form a sorted list of all elements in the ordered collection, and contain a pointer from each leaf to its successor to enable fast navigation within the sorted order. The B+-tree is also designed for collections so large that secondary storage is required. It aims to minimize the number of disk access required to both search for elements and find the set of elements that fall in a desired range of values. A B+-tree also supports efficient **bulk loading** which is important for real-time applications when a large number of insertions must be performed. If each element inserted is larger than all elements in the collection, then it can be inserted at the end of the leaf list and the rest of the structure can be built later. A B+-tree uses more space than a B-tree since the elements held in the internal nodes are duplicated in the leaves.

Other Balanced Search Trees: The AVL tree [2], a precursor to the red-black tree, stores in each node its **height**, which is the maximum length of a path (following child pointers) from that node to a leaf. The AVL tree guarantees that all methods have a logarithmic amortized cost by performing rotations whenever the height of the left and right subtree differ by more than one. The red-black tree is superior in that it can be implemented using a boolean (versus an integer) at each node to help maintain the balance, and more importantly the AVL tree could perform $O(\log n)$ rotations per insertion or deletion as opposed to at most 2 per insertion and at most 3 per deletion for the red-black tree. A 2-3-4 tree is a B-tree with $t = 2$, so each node can hold 1, 2, or 3 elements (hence 2, 3, or 4 children). A red-black tree is a way to represent a 2-3-4 tree as a binary search tree where any red node is a 2-3-4 node with its parent. Thus a red-black tree is really the best choice among these options.

SkipList: The skip list is a sorted list with additional structure that supports finding an element in expected logarithmic time. Unlike balanced search trees, a skip list achieves logarithmic performance without rearranging the structure when elements are added or removed from the collection. Furthermore, once an element is located, all other skip list methods (e.g., the insertion or removal of an element, or moving forward or backwards in the list) execute in expected constant time. The key drawback, as compared to a red-black tree, is that the search time is slower. While the probability of a search taking more than logarithmic time is extremely small, there is more variation in the search time than with a red-black tree.

Figure 29.3 shows the class hierarchy for the data structures presented in this book for the OrderedCollection ADT.

**Figure 29.3**

The class hierarchy for the OrderedCollection ADT data structures.

29.8 Further Reading

According to Knuth [97], binary search was first introduced by John Mauchly as early as 1946 and used to find the insert position for a sorted array. As Bentley [22] and Lesuisse [103] discuss, the first fully correct version of binary search did not appear until 1962.

Knuth [97] includes a good discussion of unbalanced binary search trees, which appear to have been discovered independently by several people in the late 1950s. The first form of balancing search tree, the **AVL tree** was introduced by Adel'son-Vel'skiĭ and Landis [2] in 1962. The rotations used to change the structure of the search tree (see Chapter 33) are used by all forms of balanced search trees. The key differences are the conditions used to determine when a rotation should be performed.

Bayer and McCreight [18] introduces the B-tree and also the B+-tree. Red-black trees were first proposed under the name “symmetric binary B-trees” by Bayer [17]. Guibas and Sedgewick [80] introduced the red/black color convention and studied the properties of red-black trees. An **AA-tree** is a variation of the red-black tree in which a left child may never be red [11, 156]. One can view an **ISAM index** [72], which is used commonly in database systems, as a static B+-tree in which heuristics are used to add and remove elements from the leaves.

Sleator and Tarjan [140] introduced the splay tree as a form of self-adjusting binary search tree.

Wood [158], Horowitz, Sahni, Andreson-Freed [87], and Tarjan [146] all discuss splay trees, including an analysis of the amortized complexity for them.

The expected height of a binary search tree constructed by inserting n elements in random order is roughly $2 \ln n$ [97]. Another group of ordered collection data structures introduce randomness independently of the elements themselves. As in randomized quicksort (Section 11.4.5), the expectation in the analysis for these algorithms is independent of the order in which the elements are inserted. It just depends on the randomization used within the algorithms.

A **treap** [137] is a binary search tree that also includes a *priority* for each element. (Jean Vuillemin [154] described essentially the same data structure under the name **Cartesian tree** in 1980, and McCreight [109] in 1985 introduced a **priority search tree** which is also the same basic data structure.) A treap maintains the heap ordering property that the priority of a node is never larger than that of its parent. By selecting the priority of each element uniformly at random, and then performing rotations to preserve the heap ordering property, it can be proven that the expected height of the search tree is $\Theta(\log n)$. The name treap comes from combining “tree” and “heap.”

A simpler data structure of this type is the **randomized binary search tree** [106]. An insertion into a randomized binary search tree proceeds as follows. With probability $1/(n + 1)$, where n is the number of elements currently in the ordered collection, a **root insert** is performed, which means that the new element is brought to the root after insertion using rotations (similar as to splay trees). Observe that this is the probability that a randomly selected element among $n + 1$ elements would be at the root. At each step in the standard binary search tree, a new random decision, based on the subtree size, is made as to whether or not a root insert is to be performed. If a frontier node is reached without a root insert being selected, then the standard binary search tree insertion procedure is used.

In 1990, Pugh [127] introduced skip lists. Goodrich and Tamassia [76] is another source for a discussion of skip lists. For more detailed analysis of the expected search costs of randomized skip lists, see Motwani and Raghavan [119] as well as some other papers [44, 121, 94].

Chapter 30

Sorted Array Data Structure

`AbstractCollection<E>` implements *Collection<E>*

↳ `SortedArray<E>` implements *OrderedCollection<E>*

Uses: `Array` (Chapter 11), `DynamicArray` (Chapter 13)

Used By: `B-Tree` (Chapter 36), `B+-Tree` (Chapter 37), `TaggedSortedArray` (Section 49.9.2)

Strengths: Fastest search time, as compared to other ordered collection data structures, and also very space efficient. In addition, it provides constant time access to the element in the collection at rank r . Given n elements, a sorted array can be created in $O(n \log n)$ time by using a sorting algorithm.

Weaknesses: Linear worst-case time to insert or remove elements, except for those near the maximum element of the collection.

Critical Mutators: *add* (except when the new element is larger than those currently in the collection), *addAll*, *clear*, *remove*, and *retainAll*.

Competing Data Structures: Unless the only mutations made once the ordered collection is built will be to add or remove elements near the maximum element, consider using a red-black tree (Chapter 34) or a skip list (Chapter 38). If some elements are accessed more frequently than others, or if items recently accessed are more likely to be accessed again, then a splay tree (Chapter 35) should be considered.

30.1 Internal Representation

Instance Variables and Constants: In addition to the instance variables inherited from `AbstractCollection`, a sorted array wraps a positional collection a that holds the elements. Our implementation uses an array or dynamic array for a , but either a dynamic circular array or a tracked array could be substituted. The inherited *size* instance variable is not maintained. Instead, all methods that use *size* are delegated to the wrapped positional collection.

`Array<E> a;`

Populated Example: [Figure 30.1](#) shows a populated example of a sorted array holding the letters of “ordered” inserted in the order they appear in the word.

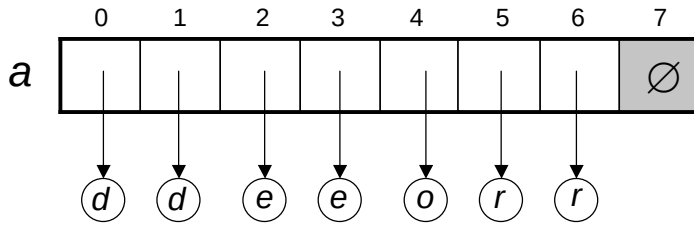


Figure 30.1

A populated example for a sorted array holding the letters of “ordered.”

Abstraction Function: Let A be a sorted array. The abstraction function

$$AF(A) = \langle u_0, u_1, \dots, u_{size-1} \rangle \text{ such that } u_p = a.get(p).$$

Design Notes: In selecting the positional collection to wrap, we have chosen to use an array if a capacity is provided, and a dynamic array otherwise. Another alternative, which would increase the number of constructors, would be to include a boolean argument to indicate whether or not the sorted array should support automatic resizing.

When an array is used, the application program can resize the array by using either *ensureCapacity* or *trimToSize*. Many applications may want to use a sorted array that supports automatic resizing, yet may still want to provide an initial capacity to prevent unnecessary resizing. Our implementation can support such applications by having the application program first call the constructor that does not take a capacity and wraps a dynamic array, and then immediately call *ensureCapacity* to resize the array.

Optimizations: In some cases, the *addAll(c)* method can be more efficiently implemented by adding all elements in c to the end of the array and then sorting the array. Loosely speaking, this alternate approach is more efficient than the one we provide when c is of comparable size (or larger than) the current collection. However, one could find the precise relationship between $|c|$ and n for which this alternate approach would be more efficient.

Also, the *set* and *add* methods that take a position and a value perform checks to ensure that the array remains sorted. To improve performance, these checks could be removed and simply require that the parameters satisfy the checked conditions.

30.2 Representation Properties

SORTED: The elements of the array are arranged in non-decreasing order. That is, $u_i \leq u_{i+1}$ for $i = 0, \dots, size-2$

Terminology: We use the following definition throughout this chapter.

- The **insert position** for *element* is any position p for which inserting *element* in position p will maintain SORTED.

Observe that an element that is *not* in the collection has a unique insert position.

30.3 Methods

In this section we present the internal and public methods for the SortedArray class.

30.3.1 Constructors

The constructor that takes no parameters creates an empty sorted array that uses the default comparator to order the elements.

```
public SortedArray() {
    this(Objects.DEFAULT_COMPARATOR);           //internally uses a dynamic array
}
```

The constructor that takes *comp*, the comparator to use to order the elements, creates a sorted array that uses the provided comparator.

```
public SortedArray(Comparator<? super E> comp){
    super(comp);
    a = new DynamicArray<E>();                 //internally use a dynamic array
}
```

The constructor that takes *capacity*, the capacity for the array, creates an empty sorted array with the given capacity. This constructor should not be used if the array should be automatically resized.

```
public SortedArray(int capacity){
    this(Objects.DEFAULT_COMPARATOR, capacity);
}
```

The constructor that takes *comp*, the user-provided comparator, and *capacity*, the capacity for the array, creates an empty sorted array of the given size that uses the provided comparator. This constructor should not be used if the array should be automatically resized.

```
public SortedArray(Comparator<? super E> comp, int capacity){
    super(comp);
    a = new Array<E>(capacity);                //internally use an array
}
```

30.3.2 Trivial Accessors

We delegate *isEmpty*, *getSize*, *getCapacity*, *ensureCapacity*, and *trimToSize* to the wrapped positional collection.

```
public boolean isEmpty(){
    return a.isEmpty();
}
public int getSize(){
    return a.getSize();
}
public int getCapacity(){
    return a.getCapacity();
}
```

```

public void ensureCapacity(int desiredCapacity){
    a.ensureCapacity(desiredCapacity);
}

public void trimToSize(){
    a.trimToSize();
}

```

30.3.3 Binary Search Algorithm

The performance of many operations depends on efficiently locating an element or computing its insert position. The binary search algorithm makes use of ORDERED to efficiently locate a desired element in a sorted array. The implementation of binary search presented here returns the insert position if the given element is not in the collection. For ease of exposition, in our discussion of the binary search algorithm, we treat $a[-1]$ as less than all elements in the domain and $a[size]$ as greater than all elements in the domain.

The *binarySearch* method takes *left*, the beginning index of the portion of the array to search, *right*, the ending index of the portion of the array to search, and *target*, the value to search for. It requires that $0 \leq left \leq right < size$ (referred to as condition P.INDEX), and that $a[left - 1] < target < a[right + 1]$ (referred to as condition P.VALUE). It returns the position where the target was found, or otherwise its insert position. Figure 30.2 illustrates the basic idea of the binary search algorithm. At each level of recursion, *target* is compared to $a[mid]$, which is the value of the middle element of the subarray $a[left], \dots, a[right]$. If $value = x$ then the target has been found. Otherwise, if $target < x$ then *target*'s position is in the portion of the array from *left* to $mid - 1$. Finally, if $target > x$ then *target*'s position is in the portion of the array from $mid + 1$ to *right*. At each step, the portion of the array that remains is reduced by half. Careful analysis reveals that at most $\lfloor \log_2(n + 1) \rfloor$ comparisons are made when the ordered collection holds n elements.

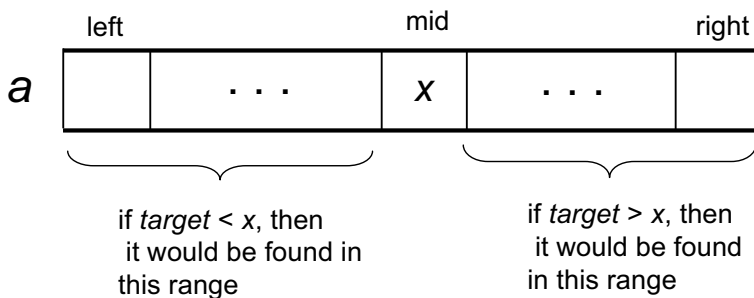


Figure 30.2

The general case for the binary search algorithm.

```

int binarySearch(int left, int right, E target) {
    int mid = (left + right)/2;           //index midway between left and right
    int comparison = comp.compare(target, a.get(mid));
    if (right == left)                    //case 1a and 1b
        return (comparison ≤ 0) ? left : right+1;
}

```

```

    else if (comparison == 0 || (comparison < 0 && left == mid))           //case 2a and 2b
        return mid;
    else if (comparison < 0)                                             //case 3
        return binarySearch(left, mid-1, target);
    else                                                                 //case 4
        return binarySearch(mid+1, right, target);
}

```

Correctness Highlights: Clearly if $a[i]$ is equivalent to $target$, then i is the proper return value. Observe that when $a[i-1] < target < a[i]$ then inserting $target$ in position i (which will move $a[i], \dots, a[n-1]$ to the right) preserves ORDERED.

We now prove that the parameters for each recursive call satisfy the requirements that

(P_INDEX)

and that $0 \leq left \leq right < size$

(P_VALUE)

$a[left - 1] < target < a[right + 1]$.

We then use these requirements to show that *binarySearch* returns the position where an equivalent was found, or otherwise the insert position for the new element.

The initial parameters values are required to satisfy the above conditions. Termination is guaranteed since the size of the subarray (i.e., $right - left + 1$) is reduced at each recursive call, and we maintain that $0 \leq left \leq right$.

We now argue inductively that the parameter requirements are maintained until termination, at which point the correct value is returned. By definition

- $comparison < 0$ is equivalent to $target < a[mid]$,
- $comparison = 0$ is equivalent to $target = a[mid]$, and
- $comparison > 0$ is equivalent to $target > a[mid]$.

Although the conditionals within the code are based on the value of $comparison$, we state the cases in the proof in terms of the relationship between $target$ and $a[mid]$. We first consider the cases when a value is returned. In these cases we use the parameter requirements to argue that the correct value is returned.

Case 1a: $left = mid = right$ and $target \leq a[mid]$. Together these conditions yield that $target \leq a[left]$. Combined with the parameter requirement, $a[left-1] < target$, it follows that $a[left-1] < target \leq a[left]$. Thus $left$ is properly returned.

Case 1b: $left = mid = right$ and $target > a[mid]$. Together these conditions yield that $target > a[right]$. Combined with the parameter requirement, $target < a[right+1]$, it follows that $a[right] < target < a[right+1]$. Thus $right + 1$ is properly returned.

Case 2a: $target = a[mid]$, so mid is properly returned.

Case 2b: $target < a[mid]$ and $left = mid$. Together these conditions yield that $target < a[left]$. Combined with the parameter requirement, $a[left-1] < target$, it follows that $a[left-1] < target < a[left]$. Thus $left$ is properly returned.

We now consider Cases 3 and 4 that make a recursive call. In these cases we inductively prove that the parameter requirements hold for the next recursive call.

Case 3: $target < a[mid]$ and $left < mid$. We first show that P_INDEX is maintained. Upon entering the method, we know that $left \leq mid - 1$ and that P_INDEX holds, so $0 \leq left \leq mid-1 < size$. Since binary search is next called with $left$ unchanged and with $mid = right + 1$, P_INDEX is maintained.

We now prove that P_VALUE is maintained. Upon entering the method we know that $target < a[mid]$ and that P_VALUE holds, so $a[left-1] < target < a[mid]$. Since binary search is next called with $left$ unchanged and with $mid = right + 1$, P_VALUE is maintained.

Case 4: $target > a[mid]$. Observe that $\lceil (left + right)/2 \rceil < right$, so $mid < right$. We first show that P_INDEX is maintained. Upon entering the method, we know that $mid + 1 \leq right$ and that P_INDEX holds, so $0 \leq mid + 1 \leq right < size$. Since binary search is next called with $right$ unchanged and with $left = mid + 1$, P_INDEX is maintained.

We now prove that P_VALUE is maintained. Upon entering the method we know that $a[mid] < target$ and that P_INDEX holds, so $a[mid] < target < a[right+1]$. Since binary search is next called with $right$ unchanged and with $left = mid + 1$, P_VALUE is maintained. ■

30.3.4 Algorithmic Accessors

The internal method *find* uses binary search to locate an element in the sorted array. The *find* method takes *element*, the target. It returns the position where an equivalent element occurs, or otherwise the insert position for *element*.

```
int find(E element){
    if (isEmpty())
        return 0;
    else
        return binarySearch(0, a.getSize()-1, element);
}
```

Correctness Highlights: For an empty collection, a new element should be inserted at slot 0. When $size \geq 1$, we need to ensure that the parameter requirements for *binarySearch* are satisfied. Substituting the parameter values $left = 0$ and $right = size - 1$ into P_INDEX gives $0 \leq 0 \leq size-1 < size$, which is satisfied when $size \geq 1$. For P_VALUE we must argue that $a[-1] < target < a[size]$ which holds by definition since we treat $a[-1]$ as smaller than all elements in the collection and $a[size]$ is larger than all elements. The remainder of the correctness argument when $size \geq 1$ follows from that of the binary search method. ■

The *contains* method takes *target*, the element being tested for membership in the collection, and returns *true* if and only if an equivalent element exists in the collection.

```
public boolean contains(E target) {
    int pos = find(target);
    return (pos < a.getSize()-1 && comp.compare(a.get(pos), target) == 0);
}
```


Correctness Highlights: For any target, the index returned by *find* is the position of an equivalent element in the collection, if any, and otherwise it is the insert position for the target. If $pos = n$, then *target* is larger than all elements in the collection, so *false* is correctly returned. Otherwise, we correctly return true exactly when $a[pos]$ is equivalent to *element*. ■

The *get* method that takes r , the desired rank, returns the r^{th} element in the sorted order, where $r = 0$ is the minimum element. It throws an *IllegalArgumentException* when $r < 0$ or $r \geq n$.

```
public E get(int r){
    if (r < 0 || r ≥ getSize())
        throw new IllegalArgumentException(“”+r);
    return a.get(r);
}
```

Correctness Highlights: By ORDERED, the element in position r has rank r . ■

The method *getEquivalentElement* takes *element*, the target, and returns an equivalent element from the collection. It throws a *NoSuchElementException* when there is no equivalent element in this collection.

```
public E getEquivalentElement(E element) {
    int pos = find(element);
    if (pos ≤ a.getSize()-1 && comp.compare(a.get(pos), element) == 0)
        return a.get(pos);
    else
        throw new NoSuchElementException();
}
```

Correctness Highlights: Like *contains* but it returns the element when *find* returns true and otherwise a *NoSuchElementException* is properly thrown. ■

The *min* method returns a smallest element in the collection. It throws a *NoSuchElementException* when the collection is empty.

```
public E min() {
    return a.get(0);
}
```

Correctness Highlights: Follows from ORDERED. ■

The *max* method returns a largest element in the collection. It throws a *NoSuchElementException* when the collection is empty.

```
public E max() {
    return a.get(a.getSize()-1);
}
```

Correctness Highlights: Follows from ORDERED. ■

To find the predecessor of an element, *find* could be used to first find some arbitrary occurrence (if any), and then iterate backwards from that occurrence until encountering a non-equivalent element,

its predecessor. However, since duplicates can occur, there could be $\Omega(n)$ occurrences of an element in which case the iteration could take linear time. Therefore we create a modification of binary search that will always return the first occurrence of the target, or the insert position if the target is not in the collection. In particular, the *findFirstInsertPosition* method takes *left*, the beginning index of the portion of the array to search, *right*, the ending index of the portion of the array to search, and *target*, the value to search for. As in *binarySearch*, this method requires that $0 \leq \text{left} \leq \text{right} < \text{size}$ and that $a[\text{left} - 1] < \text{target} < a[\text{right} + 1]$. It returns the insert position for the element that would place it before any equivalent elements.

```
protected int findFirstInsertPosition(int left, int right, E target) {
    int mid = (left + right)/2;           //index midway between left and right
    int comparison = comp.compare(target, a.get(mid));
    if (right == left)
        return (comparison ≤ 0) ? left : right+1;
    if (comparison ≤ 0) {
        if (mid == left || (comparison == 0 && comp.compare(a.get(mid-1), target) != 0))
            return mid;
        else
            return findFirstInsertPosition(left, mid-1, target);
    } else
        return findFirstInsertPosition(mid+1, right, target);
}
```

Correctness Highlights: Like that for *binarySearch*, except that when the element at position *mid* is equal to *target*, then *mid* is only returned if the element at position *mid - 1* is not equal to *target* (in which case it must be smaller by ORDERED). Otherwise, recursion proceeds with the left half of the subarray (Case 3 in the proof of binary search). ■

The *predecessor* method takes *target*, the element for which to find the predecessor. It returns the largest element in the ordered collection that is less than *target*. This method does not require that *target* be in the collection. It throws a *NoSuchElementException* when no element in the collection is smaller than *target*.

```
public E predecessor(E target) {
    if (isEmpty())
        throw new NoSuchElementException();
    int p = findFirstInsertPosition(0, a.getSize()-1, target);
    if (p == 0)
        throw new NoSuchElementException();
    else
        return a.get(p-1);
}
```

Correctness Highlights: When *target* is not in the collection, *findFirstInsertPosition* returns the position *pos* of the successor. By ORDERED, the predecessor is at position $p - 1$. Otherwise, the predecessor is at the position just before that of the first occurrence. Thus, in both cases the predecessor is at position $p - 1$. When $p = 0$, *target* is less than all elements in the collection, there is no predecessor, and the *NoSuchElementException* is properly thrown. The remainder follows from the correctness of *isEmpty*, *findFirstInsertPosition*, and the positional collection *get* method. ■

Similar to the optimization for finding the predecessor, we introduce the *findLastInsertPosition* method to efficiently find the successor when there may be duplicates. The method takes *left*, the beginning index of the portion of the array to search, *right*, the ending index of the portion of the array to search, and *target*, the value to search for. This method also requires that $0 \leq \text{left} \leq \text{right} < \text{size}$ and that $a[\text{left}-1] < \text{target} < a[\text{right}+1]$. It returns the last position where the element occurs, or otherwise the insert position.

```
protected int findLastInsertPosition(int left, int right, E target) {
    int mid = (left + right)/2;           //index midway between left and right
    int comparison = comp.compare(a.get(mid), target);
    if (comparison == 0 &&
        (mid == right || comp.compare(a.get(mid+1), target) != 0))           //Case 1
        return mid+1;
    if (right == left)                   //Case 2
        return (comparison > 0 ? left : right+1);
    if (comparison > 0)                   //Case 3
        return (mid == left ? left : findLastInsertPosition(left, mid-1, target));
    else                                  //Case 4
        return findLastInsertPosition(mid+1, right, target);
}
```

Correctness Highlights: Like that for *findFirstInsertPosition*, except that when the element at position *mid* is equal to *target*, *mid + 1* is returned only if the element at position *mid + 1* is not equal to *target* (in which case it must be larger by ORDERED). Otherwise, recursion proceeds on the right half of the subarray (Case 4). ■

The *successor* method takes *target*, the element for which to find the successor. It returns the smallest element in the ordered collection that is greater than *target*. This method does not require that *target* be in the collection. It throws a *NoSuchElementException* when no element in the collection is greater than *target*.

```
public E successor(E target) {
    if (isEmpty())
        throw new NoSuchElementException();
    int p = findLastInsertPosition(0, a.getSize()-1, target);
    if (p == a.getSize())
        throw new NoSuchElementException();
    else
        return a.get(p);
}
```

Correctness Highlights: When *target* is not in the collection, *findLastInsertPosition* returns the position *pos* of the successor. Otherwise, the successor is at the position just after that of the last occurrence. Thus, in both cases the predecessor is at position *p*. When $p = n$, *target* is greater than all elements in the collection, there is no successor, and the *NoSuchElementException* is properly thrown. The remainder follows from the correctness of *isEmpty*, *findLastInsertPosition*, and the positional collection *get* method. ■

30.3.5 Content Mutators

Methods to Perform Insertion

The *add* method takes *element*, the new element, and inserts it into the collection. When an equivalent element is already in the collection, this method adds *element* in any position that preserves ORDERED. If the application requires that new additions for equivalent elements be added after all the existing equivalent elements in the collection, such a variation can be easily created by replacing *find* by *findLastInsertPosition*. Similarly, *findFirstInsertPosition* could be used instead to insert the new element before all existing equivalent elements in the collection. Our implementation is slightly more efficient than those alternatives since the binary search can stop as soon as an equivalent element is found. We also include the optimization of checking if *element* is larger than all elements in the collection, in which case this method takes constant time.

```
public void add(E element) {
    if (a.isEmpty() || comp.compare(element, a.get(a.getSize()-1)) ≥ 0) //new element is max
        a.addLast(element);
    else
        a.add(find(element), element);
}
```

Correctness Highlights: For the sake of efficiency, if the new element is larger than the current last element (or if the collection is empty), then the element is added to the end. This clearly maintains ORDERED. Otherwise, by the correctness of *find*, the position returned is a valid insert position, so adding the new element at that position maintains ORDERED. The rest of the correctness follows from that of the positional collection *add* method. ■

Recall that the *addAll* method takes *c*, the collection to be added. This method adds all elements in *c* to the collection. Instead of adding each element of *c* individually, we add them all and then sort the array. Since quicksort runs most efficiently in practice, we use it here. If worst-case $O(n \log n)$ performance is desired, then merge sort could be used instead.

```
public void addAll(Collection<? extends E> c) {
    a.addAll(c);
    a.quicksort();
}
```

Methods to Perform Deletion

The public *remove* method takes *element*, the element to remove. It removes an arbitrary element from the collection equivalent to *element*, if such an element exists in the collection. It returns *true* if an element was removed, and *false* otherwise. Again, if desired, a variation of *remove* could be created where *find* is replaced by either *findFirstInsertPosition* (to remove the first occurrence of *element* in the iteration order), or *findLastInsertPosition* (to remove the last occurrence of *element* in the iteration order).

```
public boolean remove(E element) {
    int pos = find(element);
    if (pos == getSize() || comp.compare(element, a.get(pos)) != 0)
        return false;
    a.remove(pos);
    return true;
}
```

Recall that the *retainAll* method takes *c*, a collection, and updates the current collection to contain only elements that are also in *c*. The efficiency of this method can be substantially improved when *c* is an ordered collection. If a tracked implementation of SortedArray was used, then the call to *a.set* would need to be replaced with swap of the element at position *p* and *newSize*.

```

public void retainAll(Collection<E> c) {
    if (isEmpty())
        return;
    if (c instanceof OrderedCollection
        && c.getComparator().equals(getComparator())) {    //special case for efficiency
        int newSize = 0;                                     //number of elements retained in this collection
        int p = 0;                                           //position of next element to consider in this collection
        for (E element: c) {                                //c is sorted
            while (p < getSize() && comp.compare(element, get(p)) > 0)
                p++;
            while (p < getSize() && comp.compare(element, get(p++)) == 0)
                a.set(newSize++, element);                  //if tracked instead use a swap
        }
        a.removeRange(newSize, getSize()-1);
    }
    else
        a.retainAll(c);
}

```

Correctness Highlights: The collection does not change when it is empty. We now consider when the collection is not empty.

First we consider when *c* is an ordered collection. We maintain the loop invariant that the elements in positions $0, \dots, \text{newSize} - 1$ of *a* are in *c*, and that their relative order is maintained from *a*, and that all elements in positions *newSize*, $\dots, p - 1$ are not in *c*. Initially, this invariant holds since *newSize* = *p* = 0. The for loop iterates through all elements in *c*. The element of *a* in position *p* is being considered. By SORTED, if *element* is greater than the element in this collection at position *p*, then *element* is not in *c* so it should not be retained. So incrementing *p* preserves the invariant. If *element* is equivalent to the element in this collection at position *p* then that element is moved to position *newSize*, and both *newSize* and *p* are incremented to preserve the invariant. Finally, when *element* is smaller than the element in this collection at position *p*, the for loop advances to the next element in *c*. Upon termination of the for loop, by the invariant it follows that the elements in positions $0, \dots, \text{newSize} - 1$ should remain in *a* at their current position, and those in positions *newSize*, $\dots, n - 1$ should be removed. The rest of the correctness follows from that of *removeRange*.

When *c* is not an ordered collection, the correctness follows from that of the abstract collection *retainAll* method. ■

30.3.6 Utilities for the B-Tree and B+-Tree Classes

Some data structures internally use a sorted array. For example, the B-tree data structure presented in Chapter 36 uses a sorted array within each of its nodes. To support such data structures, we provide methods to access slots of the sorted array directly by index.

Since the B-tree and B+-tree algorithms compute the position for the new element, it would be redundant and wasteful to incur logarithmic search cost. Therefore, we provide a *set* method

that takes *pos*, the desired position, and *value*, the value to be added at position *pos*. Since this method bypasses the search for the insertion position, it must guard against uses that would violate ORDERED. It throws a *PositionOutOfBoundsException* when *pos* is not a valid position in *a*. It throws an *IllegalArgumentException* when performing this operation would violate ORDERED. To improve performance, one could remove these checks and simply require that the parameters satisfy the conditions.

```
protected void set(int pos, E value){
    if ((pos > 0) && comp.compare(value, a.get(pos-1)) < 0)
        throw new IllegalArgumentException("value " + value +
            "is smaller than the element at position " + (pos-1));
    if ((pos < a.getSize()-1) && comp.compare(value, a.get(pos+1)) > 0)
        throw new IllegalArgumentException("value " + value +
            " is larger than the element at position " + (pos+1));
    a.set(pos, value);
}
```

Correctness Highlights: An exception is thrown if ORDERED would be violated. The rest follows from the correctness of the positional collection *set* method. ■

The *add* method takes *pos*, the desired position, and *element*, the element to be added at position *pos*. It throws a *PositionOutOfBoundsException* when *pos* is not a valid position for adding to *a*. It throws an *IllegalArgumentException* when performing this operation would violate ORDERED. As discussed for *set*, to improve performance, one could remove these checks and simply require that the parameters satisfy the conditions.

```
protected void add(int pos, E element){
    if ((pos > 0) && comp.compare(element, a.get(pos-1)) < 0)
        throw new IllegalArgumentException("value " + element +
            "is smaller than the element at position " + (pos-1));
    if ((pos < a.getSize()-1) && comp.compare(element, a.get(pos)) > 0)
        throw new IllegalArgumentException("value " + element +
            " is larger than the element currently at position " + pos);
    a.add(pos, element);
}
```

Correctness Highlights: An exception is thrown if ORDERED would be violated. The rest follows from the correctness of the positional collection *add* method. ■

The *remove* method takes *pos*, the position of the element to removed. It returns the removed element. It throws a *PositionOutOfBoundsException* when *p* is not a valid position in *a*.

```
protected E remove(int pos){
    return a.remove(pos);
}
```

Correctness Highlights: Removing an element cannot violate ORDERED. The rest follows from the correctness of the positional collection *remove* method. ■

30.3.7 Locator Initializers

These methods are all delegated to the positional collection class. Note that a `BasicMarker` is used, rather than the marker that the iterator methods of the array would normally return. This is to prevent uses from calling methods like *addAfter* that could violate `ORDERED`.

```
public Locator<E> iterator() {
    return a.new BasicMarker(Array.FORE);
}

public Locator<E> iteratorAtEnd() {
    return a.new BasicMarker(a.getSize());
}

public Locator<E> getLocator(E element) {
    int pos = find(element);
    if (pos ≥ a.getSize() || comp.compare(a.get(pos), element) != 0)
        throw new NoSuchElementException();
    else
        return a.new BasicMarker(pos);
}
```

Correctness Highlights: By the correctness of *find*, if *element* is in the collection then it is at position *pos*. Furthermore, $0 \leq pos \leq size$. If $pos = size$ then the element is not in the collection. The rest of the correctness follows from that of the array *getSize*, *get*, and the `BasicMarker` class. ■

30.4 Performance Analysis

The asymptotic time complexities of all public methods for the `SortedArray` class are shown in [Table 30.3](#). Observe that *iterator* and *iteratorAtEnd* take constant time since the array and dynamic array locator constructors take constant time. Since the *minimum* element is known to be at position 0, and the *maximum* element is known to be at position $n - 1$, both of these methods have constant cost.

The time complexity for *contains*, *getEquivalentElement*, *getLocator*, *predecessor*, and *successor* are dominated by the binary search (or its variants). At each step, the size of the subarray under consideration (i.e., $right - left + 1$) is reduced by a factor of 2. More specifically, observe that if $n = 2^x - 1$ for an integer x , then exactly x comparisons are made. In general, the number of comparisons is the smallest value of y for which $2^y - 1 \geq n$. Therefore, when there are n elements in the collection, at most $\lceil \log_2(n + 1) \rceil$ comparisons are made, and that is the dominant part of the computation. Thus *contains*, *getEquivalentElement*, *predecessor*, and *successor* take logarithmic time.

The *ensureCapacity* and *trimToSize* methods are delegated to the positional collection. By `SORTED`, the iteration order corresponds to the order the elements are stored in the array. Thus the *accept*, *clear*, *toArray*, and *toString* methods take linear time since they spend constant time for each element when iterating through the collection.

For both *add* and *remove*, the position for adding or removing an element can be found in $O(\log n)$ time using binary search, but in the worst case it takes $O(n)$ time to add or remove an

method	time complexity
<i>iterator()</i>	$O(1)$
<i>iteratorAtEnd()</i>	$O(1)$
<i>max()</i>	$O(1)$
<i>min()</i>	$O(1)$
<i>contains(o)</i>	$O(\log n)$
<i>getEquivalentElement(o)</i>	$O(\log n)$
<i>getLocator(o)</i>	$O(\log n)$
<i>predecessor(o)</i>	$O(\log n)$
<i>successor(o)</i>	$O(\log n)$
<i>accept(v)</i>	$O(n)$
<i>add(o)</i>	$O(n)$
<i>clear()</i>	$O(n)$
<i>remove(o)</i>	$O(n)$
<i>toArray()</i>	$O(n)$
<i>toString()</i>	$O(n)$
<i>trimToSize()</i>	$O(n)$
<i>retainAll(OrderedCollection c)</i>	$O(n + c)$
<i>ensureCapacity(x)</i>	$O(x)$
<i>addAll(c)</i>	$O((n + c) \log(n + c))$
<i>retainAll(c)</i>	$O(n c + n)$

Table 30.3 Summary of the asymptotic time complexities for the public methods when using a sorted array to implement the OrderedCollection ADT.

locator method	time complexity
constructor	$O(1)$
<i>advance()</i>	$O(1)$
<i>get()</i>	$O(1)$
<i>hasNext()</i>	$O(1)$
<i>inCollection()</i>	$O(1)$
<i>next()</i>	$O(1)$
<i>retreat()</i>	$O(1)$
<i>remove()</i>	$O(n)$

Table 30.4 Summary of the time complexities for the locator methods for the sorted array.

element from an array. Thus, both of these methods have worst-case linear cost. The *addAll* method adds all elements to the collection, resulting in a collection of size $n + |c|$. Then quicksort is used to restore SORTED, so the expected time complexity is $O((n + |c|) \log(n + |c|))$. If worst case $O((n + |c|) \log(n + |c|))$ cost is desired, merge sort could be used instead.

When *retainAll* is called with a collection that is not an ordered collection, it must spend $O(|c|)$ to search for all n elements in c , and then $O(n)$ time is spent for each element removed. Since in the worst case all n elements are removed, the time complexity is $O(n(|c| + n))$. When c is an ordered collection, then *retainAll* simultaneously iterates through c and n , just retaining the elements that are in c . So, when c is an ordered collection, *retainAll* has time complexity $O(n + |c|)$.

Since the sorted array locator is the positional collection locator, the time complexities for all of the public methods performed using the locator are the same as those for the Array locator. These time complexities are shown in Table 30.4. Recall that *advance* moves to the next element in the iteration order (even if it has an equal value). Thus, *advance* followed by *get* could either return the successor of the current element or an equal element. In summary, the locator methods all take constant time, except that *remove* takes worst-case linear time due to the possible need to shift up to half of the array elements.

30.5 Quick Method Reference

SortedArray Public Methods

- p. 457 *SortedArray*()
- p. 457 *SortedArray*(Comparator<? super E> comp)
- p. 457 *SortedArray*(Comparator<? super E> comp, int capacity)
- p. 457 *SortedArray*(int capacity)
- p. 98 void **accept**(Visitor<? super E> v)
- p. 464 void **add**(E element)
- p. 100 void **addAll**(Collection<? extends E> c)
- p. 100 void **clear**()
- p. 97 boolean **contains**(E value)
- p. 99 void **ensureCapacity**(int capacity)
- p. 461 E **get**(int r)
- p. 96 int **getCapacity**()
- p. 96 Comparator<? super E> **getComparator**()
- p. 98 E **getEquivalentElement** (E target)
- p. 467 Locator<E> **getLocator**(E element)
- p. 96 int **getSize**()
- p. 96 boolean **isEmpty**()
- p. 467 Locator<E> **iterator**()
- p. 467 Locator<E> **iteratorAtEnd**()
- p. 461 E **max**()
- p. 461 E **min**()
- p. 462 E **predecessor**(E target)
- p. 464 boolean **remove**(E element)
- p. 100 void **retainAll**(Collection<E> c)
- p. 463 E **successor**(E target)
- p. 97 Object[] **toArray**()
- p. 97 E[] **toArray**(E[] array)
- p. 98 String **toString**()
- p. 99 void **trimToSize**()

SortedArray Internal Methods

- p. 466** void **add**(int pos, E element)
- p. 458** int **binarySearch**(int left, int right, E target)
- p. 97 int **compare**(E e1, E e2)
- p. 97 boolean **equivalent**(E e1, E e2)
- p. 460** int **find**(E element)
- p. 462** int **findFirstInsertPosition**(int left, int right, E target)
- p. 463** int **findLastInsertPosition**(int left, int right, E target)
- p. 466** E **remove**(int pos)
- p. 466** void **set**(int pos, E value)
- p. 99 void **traverseForVisitor**(Visitor<? super E> v)
- p. 98 void **writeElements**(StringBuilder s)

Chapter 31

Abstract Search Tree Class

AbstractCollection<E> implements *Collection*

↳ AbstractSearchTree<E> implements *Collection*<E>

31.1 Internal Representation

An abstract search tree is a rooted tree (see Section 1.6), for which each tree node holds a sequence of elements $e_0, \dots, e_{s-1}$ where s is the size of the tree node. A tree node holding s elements will have $s + 1$ child references $C_0, \dots, C_s$. Each tree node has an instance variable, *parent*, that references its parent, where the root is distinguished by having a *null* parent.

Element e_i can be viewed as having a left child C_i and a right child C_{i+1} . For example, in the search tree shown in Figure 31.1, for the node labeled “c f,” $e_0 = \text{“c,”}$ $e_1 = \text{“f.”}$ C_0 refers to the node labeled “a b,” C_1 refers to the node labeled “d e,” and C_2 refers to the node labeled “g h.” Element “f” (e_1), has left child “d e” (C_1) and right child “g h” (C_2).

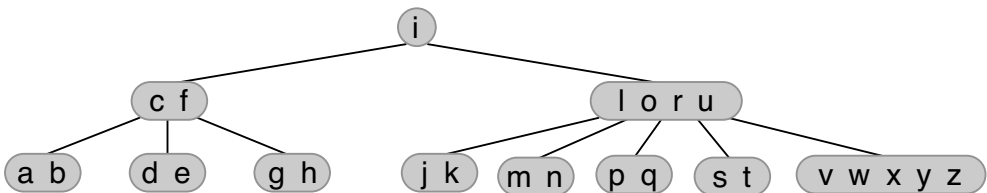


Figure 31.1

A search tree holding an ordered collection of the letters of the alphabet.

The **subtree** rooted at node x includes x and all other nodes that can be reached from x by following child references. The main property of a search tree is that $e_0 \leq e_1 \leq \dots \leq e_{s-1}$ and all elements in the subtree rooted at C_i are less than or equal to e_i , and all elements in the subtree rooted at C_{i+1} are greater than or equal to e_i . For example, in the search tree shown in Figure 31.1, all elements in the subtree rooted at “c f” are less than “i” and all elements in the subtree rooted at “l o r u” are greater than “i.” Furthermore, the node labeled “l o r u” (with $s = 4$) illustrates how the letters from j to z are partitioned into 5 children based on where they fall in sorted order with respect to l, o, r , and u . This property enables an element to be found in a single pass down the tree and also allows an inorder traversal, shown in the implementation of *traverseForVisitor*, to be used to visit the elements from the collection in sorted order in time proportional to the size of the collection.

Terminology: We use the following definitions throughout our discussion of all data structures that extend `AbstractSearchTree`. In all of these definitions we use x to denote a tree node.

- Let $s = x.size()$ denote the number of elements in x . The elements held x are $x.data(0), \dots, x.data(s-1)$, and the children are $x.child(0), \dots, x.child(s)$.
- For element $x.data(i)$, we refer to $x.child(i)$ as its **left child** and $x.child(i+1)$ as its **right child**.
- We refer to the sibling of x just to its left, if it exists, as its **left sibling**, and the sibling just to its right, if it exists, as its **right sibling**. More specifically, p be the parent of x , where $x = p.child(i)$. The left sibling of x is $p.child(i-1)$ and the right sibling of x is $p.child(i+1)$.
- For an element e and a collection S of elements, we say that $e \leq S$ if and only if for all $e' \in S$, $e \leq e'$. Likewise, we say that $e \geq S$ if and only if for all $e' \in S$, $e \geq e'$.
- A **frontier** node is a tree node that corresponds to an empty collection. For example, in Figure 31.1, the tree node holding “ab” has three children that are frontier nodes. When drawing a search tree, we do not show the frontier nodes. Any child pointer not shown by convention references a frontier node. While *null* could be used as an alternative, using frontier nodes removes special cases within the code. As discussed in much more depth in Section 32.1, we do not create a new frontier node at each *null* pointer. Instead, we introduce two shared frontier nodes to reduce the space used.
- We say that a tree node w is a **descendant** of x when either $w = x$, or w can be reached through following a sequence of child references from x .
- An **ancestor** of x is any tree node reachable from x by following the parent pointers from x .
- We define the **subtree rooted** at x to be the portion of the tree that includes all descendants of x .
- We let $T(x)$ denote the set of non-frontier nodes in the subtree rooted at x .
- We say that x is in the collection if and only if $x \in T(root)$.
- A **leaf** is a non-frontier node whose children are only frontier nodes. A leaf is sometimes called an **external node**.
- An **internal node** is any non-frontier node that is not a leaf node. That is, an internal node has some child that is not a frontier node.
- We define the **search path** for e to be the sequence of tree nodes that are followed in a search for e that begins at the root, and follows the appropriate child pointer until either e is found or a frontier node is reached. For example, in the search tree shown in Figure 31.1, the search path for “e” begins at the root node, then goes to the node labeled “c f,” and then to the node labeled “d e.” The search path for an element between “e” and “f” would start at the root, proceed to the node labeled “c f,” continue to the node labeled “d e” and finally reach the frontier node referenced by C_2 of the node labeled “d e.”
- The **insert position** for an element e is the location in the search tree where e would be inserted. For ease of implementation, we represent the insert position as a reference to a frontier node whose *parent* variable refers to the node preceding the frontier node on the search path for e .
- The **leftmost node** in $T(x)$ is the leaf node obtained by following the leftmost child reference (C_0) starting at x , stopping at the node whose child reference C_0 refers to a frontier node.

- The **rightmost node** in $T(x)$ is defined as the leaf node obtained by following the rightmost child reference (C_s) starting at x , stopping at the node whose child reference C_s refers to a frontier node.
- We let “ $\langle \rangle$ ” denote an empty sequence, and let “ $+$ ” denote the concatenation operator when applied to a sequence, and let $\sum_{i=0}^{s-1} e_i$ denote $e_0 + \dots + e_{s-1}$.
- The **iteration order** given by $seq(root)$ is defined recursively as

$$seq(x) = \begin{cases} \langle \rangle, & \text{if } x \text{ is a frontier node} \\ \sum_{i=0}^{s-1} [seq(x.child(i)) + x.data(i)] + seq(x.child(s)) & \text{otherwise} \end{cases}$$

where $s = x.size()$. In the correctness argument for *traverseForVisitor*, we prove that $seq(x)$ contains the elements in $T(x)$ in sorted order. For now, we just illustrate this fact using an example. Let *root* be a reference to the root of the search tree shown in Figure 31.1, let *left* refer to the tree node labeled “c f,” and let *right* refer to the tree node labeled “l o r u.” Applying the recursive definition about yields that $seq(root) = seq(left) + “i” + seq(right)$. Similarly, we get that $seq(left) = “a b” + “c” + “d e” + “f” + “g h” = “a b c d e f g h”$ where we apply the observation given above. Likewise, $seq(right) = “j k l m n o p q r s t u v w x y z.”$ Thus $seq(root) = “a b c d e f g h i j k l m n o p q r s t u v w x y z.”$

- We define the **predecessor** of element e to be the element that comes immediately before it in the iteration order. By definition, the first element in the sequence has no predecessor.
- We define the **successor** of element e to be the element that comes immediately after it in the iteration order. By definition, the last element in the sequence has no successor.

Abstraction Function: Let ST be a search tree instance. The abstraction function

$$AF(ST) = seq(root).$$

31.2 Representation Properties

We inherit SIZE and introduce the following additional representation properties. INORDER is the key to providing logarithmic search time, as well as a linear time method to traverse through the elements in the collection in sorted order. REACHABLE defines which tree nodes hold elements in the collection. PARENT states that, with the exception of frontier nodes, the parent and child pointers are consistent. That is, if node y is a child of node x , then $y.parent = x$. Finally, FRONTIER captures the way we represent an empty child pointer by referencing a frontier node.

INORDER: For all reachable nodes x , if $s = x.size()$,

$$T(x.child(0)) \leq x.data(0) \leq T(x.child(1)) \leq \dots \leq x.data(s-1) \leq T(x.child(s)).$$

REACHABLE: The elements held within the tree nodes in $T(root)$ are exactly those in the collection.

PARENT: For all reachable nodes x , either $x.child(i)$ is a frontier node or $x.child(i).parent = x$. Finally, $root.parent = null$.

FRONTIER: For all reachable nodes x , when $T(x.child(i))$ is empty, $x.child(i)$ references a frontier node.

31.3 Abstract Tree Node Inner Class

In this section, we describe the methods that must be supported by the `TreeNode` inner class.

The abstract method *size* returns the number of elements held in that tree node.

protected abstract int size();

The abstract method *capacity* returns the maximum number of elements that are allowed in the tree node.

protected abstract int capacity();

The abstract method *child* takes *index*, the index for the desired child, and returns the tree node reference for that child. That is, C_{index} is returned.

protected abstract `TreeNode` child(**int** index);

The method *leftmostChild* returns the leftmost child.

```
final TreeNode leftmostChild(){
    return child(0);
}
```

The method *rightmostChild* returns the rightmost child.

```
final TreeNode rightmostChild(){
    return child(size());
}
```

The abstract method *data* takes *index*, the index of the desired element, and returns the element. That is, e_{index} is returned. It throws an *IllegalArgumentException* when $index < 0$ or $index \geq size()$.

protected abstract `E` data(**int** index);

The abstract method *isFrontier* returns *true* if and only if the node is the frontier node.

protected abstract boolean isFrontier();

31.4 Abstract Search Tree Class

We now present the implementation of the abstract search tree class. This abstract class forms the basis for the implementation of the `BinarySearchTree`, `RedBlackTree`, `SplayTree`, `BTree`, and `BPlusTree` classes.

Instance Variables and Constants: The variables *size*, *comp*, *DEFAULT_CAPACITY*, and *version* are inherited from `AbstractCollection`. The only variable added for the `AbstractSearchTree` class is *root*, a reference to the root of the search tree.

```
protected TreeNode root;
```

31.5 Abstract Search Tree Methods

In this section we present internal and public methods for the `AbstractSearchTree` class.

31.5.1 Constructors

The constructor that takes no parameters creates an empty search tree that uses the default comparator.

```
public AbstractSearchTree() {
    this(Objects.DEFAULT_COMPARATOR);
}
```

The constructor that takes *comp*, the element comparator, creates an empty search tree that uses the given comparator.

```
public AbstractSearchTree(Comparator<? super E> comp) {
    super(comp);
}
```

31.5.2 Algorithmic Accessors

The abstract method *find* takes *target*, the element to be located. It returns a reference to a tree node that contains an occurrence of the element if it is found. If *target* is not in the collection, then it returns the frontier node at the insert position. In that case, the parent of the returned frontier node must be the node that preceded it on the search path. This requirement makes it possible to retrace the path from the frontier node to the root. As discussed in Section 32.1, the frontier node may be shared for efficiency. In such cases, the *find* call and the subsequent insertion must occur atomically with respect to other operations on the collection.

```
protected abstract TreeNode find(E target);
```

The *contains* method takes *target*, the element being tested for membership in the collection, and returns *true* if and only if an equivalent value exists in the collection.

```

public boolean contains(E target) {
    return (!find(target).isFrontier());
}

```

Correctness Highlights: By the correctness of *find*, the desired element is in the collection exactly when the value returned is not a frontier node. ■

The method *get* takes *r*, the desired rank. It returns the r^{th} element in the sorted order, where $r = 0$ is the minimum. It throws an *IllegalArgumentException* when $r < 0$ or $r \geq n$.

```

public E get(int r) {
    if (r < 0 || r ≥ getSize())
        throw new IllegalArgumentException();
    Locator<E> loc = iterator();
    for (int j=0; j < r+1; j++, loc.advance());
    return loc.get();
}

```

Correctness Highlights: By the correctness of the locator methods, and the fact that the iteration order is based on the sorted order, advancing $r + 1$ times from FORE leaves the locator at the rank *r* element. ■

The internal method *getLastNodeSearchIndex* returns the index, within its tree node, of the element returned in the most recent search. This method requires that it is called only after a successful search.

```

protected int getLastNodeSearchIndex() {
    return 0;
}

```

Correctness Highlights: For a binary search tree in which there is only one element in each tree node, the element found during the last successful search must be for the element at index 0. ■

The *getEquivalentElement* method takes *target*, the target element, and returns an equivalent element that is in the collection. It throws a *NoSuchElementException* when there is no equivalent element in the collection.

```

public E getEquivalentElement(E target) {
    TreeNode t = find(target);
    if (t.isFrontier())
        throw new NoSuchElementException();
    return t.data(getLastNodeSearchIndex());
}

```

Correctness Highlights: By the correctness of *find*, the desired element is in the collection exactly when the value returned is not a frontier node. The rest of the correctness follows from that of *getLastNodeSearchIndex* and the tree node *data* method. ■

The internal method *leftmost* takes *x*, a reference to any tree node in the collection. It returns the leftmost tree node in the subtree rooted at *x*.


```

TreeNode leftmost(TreeNode x) {
    while (!x.leftmostChild().isFrontier())
        x = x.leftmostChild();
    return x;
}

```

Correctness Highlights: Termination is guaranteed by FRONTIER. The rest of the correctness from that of *leftmostChild*, *isFrontier*, and the definition of the leftmost node in $T(x)$. ■

The *leftmost* method without any parameters returns the leftmost node in the tree.

```

TreeNode leftmost() {
    return leftmost(root);
}

```

Correctness Highlights: Follows from REACHABLE and the correctness of the *leftmost* method that takes a tree node as a parameter. ■

The *min* method returns a smallest element in the collection. It throws a *NoSuchElementException* when the collection is empty.

```

public E min() {
    if (isEmpty())
        throw new NoSuchElementException();
    else
        return leftmost().data(0);
}

```

Correctness Highlights: When the collection is not empty, we know by INORDER, that if the leftmost node is x , then $x.data(0)$ is no greater than all other elements in the collection. The rest follows from the correctness of *leftmost*. ■

The internal method *rightmost* takes x , a reference to any tree node in the collection. It returns the rightmost tree node in the subtree rooted at x .

```

TreeNode rightmost(TreeNode x) {
    while (!x.rightmostChild().isFrontier())
        x = x.rightmostChild();
    return x;
}

```

Correctness Highlights: Termination is guaranteed by FRONTIER. The rest of the correctness follows by the correctness of *rightmostChild*, *isFrontier*, and the definition of the rightmost node in $T(x)$. ■

The *rightmost* method without any parameters returns the rightmost node in the tree.

```

TreeNode rightmost() {
    return rightmost(root);
}

```

The *max* method returns a greatest element in the collection. It throws a *NoSuchElementException* when the collection is empty.

```
public E max() {
    if (isEmpty())
        throw new NoSuchElementException();
    else {
        TreeNode x = rightmost();
        return x.data(x.size()-1);
    }
}
```

Correctness Highlights: When the collection is not empty, we know by INORDER, that if the rightmost node is x , then $x.data(x.size - 1)$ is no smaller than all other elements in the collection. The rest follows from the correctness of *rightmost*. ■

INORDER is crucial to both efficiently locate an element and to perform an inorder traversal to visit the elements from the collection in sorted order in linear time. An **inorder traversal** is a recursive procedure based on the recursive definition of $seq(x)$ that will visit the elements in sorted order.

The *traverseForVisitor* method that takes v , the visitor, and x , a reference to the tree node, follows the recursive definition given for $seq(x)$. Any exception thrown by the visitor propagates to the calling method. Although the inherited abstract collection method *traverseForVisitor* that uses an iterator would work here, we override it to perform the traversal more efficiently.

```
void traverseForVisitor(Visitor<? super E> v, TreeNode x) throws Exception {
    if (!x.isFrontier()) {
        for (int i = 0; i < x.size(); i++){
            traverseForVisitor(v, x.child(i));
            v.visit(x.data(i));
        }
        traverseForVisitor(v, x.rightmostChild());
    }
}
```

Correctness Highlights: By construction, this method visits the elements in the iteration order (as defined by $seq(x)$). We now argue that the iteration order is sorted in a non-decreasing order. By INORDER it follows that for each recursive call made to this method, all elements in $x.child(i)$ are less than or equal to the element in $x.data(i)$. Similarly, all elements in $x.rightmostChild()$ (which is $x.child(x.size())$) are greater than the element $x.data(x.size() - 1)$. By construction, all elements in the subtree rooted at x are visited. Termination is guaranteed by FRONTIER. ■

The method *traverseForVisitor* takes v , the visitor, and applies the visitor to each element of the collection in sorted order.

```
protected void traverseForVisitor(Visitor<? super E> v) throws Exception {
    traverseForVisitor(v, root);
}
```

Correctness Highlights: By REACHABLE and the correctness of the other *traverseForVisitor* method, all vertices are visited in the iteration order. ■

Recall that *writeElements* takes *sb*, a string builder, and appends to the string buffer a comma-separated string representation for the elements in the collection, in the iteration order. For the sake of efficiency, *StringBuilder* is used instead of *String*. If a *String* were directly used, then this method (and consequently the collection's *toString* method) would take quadratic time since the accumulated string would be copied for each append. To perform the traversal more efficiently, as with *traverseForVisitor*, the implementation uses a visitor, which visits the elements of the collection using an inorder traversal.

```
protected void writeElements(final StringBuilder sb) {
    if (!isEmpty()) {                                     //only visit the collection if it is not empty
        accept(new Visitor<E>() {
            public void visit(E item) throws Exception {
                sb.append(item);
                sb.append(", ");
            }
        });
        int extraComma = sb.lastIndexOf(", "); //remove the comma and space characters
        sb.delete(extraComma, extraComma+2);    //after the last element
    }
}
```

31.5.3 Content Mutators

The abstract internal *insert* method takes *element*, the new element, and inserts it into the collection. It returns a reference to the newly inserted element.

```
protected abstract TreeNode insert(E element);
```

The *add* method takes *element*, the new element, and inserts *element* into the collection.

```
public void add(E element) {
    insert(element);
    size++;
}
```

Correctness Highlights: This method maintains SIZE. The remainder of the correctness follows from that of *insert*. ■

The abstract internal *remove* method takes *ptr*, a pointer to an existing tree node, and removes it.

```
protected abstract void remove(TreeNode ptr);
```

Finally, the public *remove* method takes *element*, the element to be removed, and removes an arbitrary element in the collection equivalent to *element*, if any. It returns *true* if and only if an element is removed.

```

public boolean remove(E element) {
    TreeNode ptr = find(element);
    if (ptr.isFrontier())
        return false;
    remove(ptr);
    return true;
}

```

31.6 Quick Method Reference

AbstractSearchTree Public Methods

- p. 475 *AbstractSearchTree*()
- p. 475 *AbstractSearchTree*(Comparator<? super E> comp)
- p. 98 void **accept**(Visitor<? super E> v)
- p. 479 void **add**(E element)
- p. 100 void **addAll**(Collection<? extends E> c)
- p. 100 void **clear**()
- p. 97 boolean **contains**(E value)
- p. 99 void **ensureCapacity**(int capacity)
- p. 476 E **get**(int r)
- p. 96 int **getCapacity**()
- p. 96 Comparator<? super E> **getComparator**()
- p. 98 E **getEquivalentElement** (E target)
- p. 96 int **getSize**()
- p. 96 boolean **isEmpty**()
- p. 478 E **max**()
- p. 477 E **min**()
- p. 479 boolean **remove**(E element)
- p. 100 void **retainAll**(Collection<E> c)
- p. 97 Object[] **toArray**()
- p. 97 E[] **toArray**(E[] array)
- p. 98 String **toString**()
- p. 99 void **trimToSize**()

AbstractSearchTree Internal Methods

- p. 97 int **compare**(E e1, E e2)
- p. 97 boolean **equivalent**(E e1, E e2)
- p. 476 int **getLastNodeSearchIndex**()
- p. 477 TreeNode **leftmost**()
- p. 476 TreeNode **leftmost**(TreeNode x)
- p. 477 TreeNode **rightmost**()
- p. 477 TreeNode **rightmost**(TreeNode x)
- p. 99 void **traverseForVisitor**(Visitor<? super E> v)
- p. 478 void **traverseForVisitor**(Visitor<? super E> v, TreeNode x)
- p. 98 void **writeElements**(StringBuilder s)
- p. 479 void **writeElements**(StringBuilder sb)

Chapter 32

Binary Search Tree Data Structure

AbstractCollection<E> implements *Collection*<E>

└ AbstractSearchTree<E> implements *OrderedCollection*<E>

└ BinarySearchTree<E> implements *OrderedCollection*<E>, *Tracked*<E>

Uses: Java references

Used By: KDTreeImpl (Section 47.5), TaggedBinarySearchTree (Section 49.9.3)

Strengths: When the elements are inserted in a random order (i.e., in the order of a random permutation of the elements in the ordered collection), then good performance is obtained. No computation time is consumed to balance the search tree.

Weaknesses: If the elements are inserted in sorted order, or in reverse sorted order, then the tree degenerates into a sorted list with linear (versus logarithmic) time to locate an element. In fact, very unbalanced trees can occur in practice unless the insertion order is close to random.

Critical Mutators: none

Competing Data Structures: A sorted array (Chapter 30) is preferable if fast search time is very important and elements added or removed are near the maximum, or when there are no mutations after inserting all elements. Also a sorted array is a good choice if space usage is to be minimized, or if constant time access is needed for retrieving the element at rank i . If elements are not guaranteed to be inserted in random order (or if deletions are common), then some form of a balanced binary search tree (Chapter 33) or a skip list (Chapter 38) is preferable to a binary search tree. If the collection is so large that secondary storage will be required, then either a B-tree (Chapter 36) or a B+-tree (Chapter 37) should be considered.

32.1 Internal Representation

A binary search tree is a special form of an abstract search tree, in which each tree node holds one element and has two children. We refer to *child(0)* as the **left child** and *child(1)* as the **right child**. (See [Figure 32.1](#).) Since each node in the search tree holds a single element from the collection, our discussion treats the element its containing node interchangeably. For an element e , we refer to the portion of the binary search tree referenced by e 's left child as e 's **left subtree**. Similarly, we refer to the portion of the binary search tree referenced by e 's right child as e 's **right subtree**. For

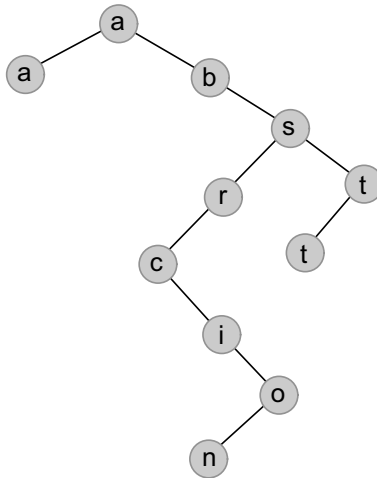


Figure 32.1

A populated example of a binary search tree holding the letters of “abstraction,” inserted in the order they appear in the word. All left child pointers not shown reference *FRONTIER_L* and all right child pointers not shown reference *FRONTIER_R*. The root is the “a” shown at the top of the figure, with a left child of “a” and a right child of “b.”

a binary search tree, INORDER implies that for every element e , all elements in e ’s left subtree are less than or equal to e , and all elements in e ’s right subtree are greater than or equal to e .

Instance Variables and Constants: The following instance variables and constants are defined for the `BinarySearchTree` class in addition to those inherited from the `AbstractSearchTree` class.

```

protected final BSTNode FRONTIER_L = createFrontierNode();
protected final BSTNode FRONTIER_R = createFrontierNode();

```

We use the frontier nodes to eliminate special cases in the code. The constant *FORE* is a tree node that is logically just before the first element in the collection, and the constant *AFT* is a tree node that is logically just after the last element in the collection.

```

final BSTNode FORE = createTreeNode(null);
final BSTNode AFT = createTreeNode(null);

```

Populated Example: Figures 32.1 and 32.2 show populated examples of binary search trees.

Terminology: We use terminology introduced for the abstract search tree (Chapter 31). Observe that for a binary search tree, we can use the fact that each node has a single element and at most two children to provided a simplified, equivalent, definition for $seq(x)$:

$$seq(x) = \begin{cases} \langle \rangle, & \text{if } x \text{ is a frontier node} \\ seq(x.left) + x.data + seq(x.right) & \text{otherwise.} \end{cases}$$

We introduce the following additional definitions.

- The **internal iteration order** is the order defined by an inorder traversal over the conceptual tree, where the $n + 1$ frontier nodes are included. For the binary search tree of Figure 32.2,

the internal insertion order is $\langle f_1, \text{"d," } f_2, \text{"e," } f_3, \text{"o," } f_4, \text{"r," } f_5, \text{"r," } f_6 \rangle$ where the first "r" shown is the one with children f_4 and f_5 .

Recall that, for an element that is *not* in the collection, our definition of the insert position identifies a unique frontier node for the placement of that element. For an element that *is* in the collection, if m is the number of occurrences of the element, then there are $m + 1$ possible insert positions surrounding the m occurrences of the element. Inserting the element at any of these would satisfy INORDER, but we identify two particular insert positions, the first and the last, primarily to assist in defining the methods *predecessor* and *successor*, respectively.

- The **first insert position** for element e not in the collection is the unique insert position. For an element e in the collection, the **first insert position** is the frontier node in the internal iteration order that immediately precedes the first occurrence of e . For example, the first insert position for "r" is at frontier node f_4 .
- The **last insert position** for element e not in the collection is the unique insert position. For an element e not in the collection, the **last insert position** is the frontier node in the internal iteration order that immediately follows the last occurrence of e . For example, the last insert position for "r" is at frontier node f_6 .
- For a node x holding element e , we define the **predecessor** of x to be the node that holds e 's predecessor. Similarly, we define the **successor** of x to be the node that holds e 's successor.
- We say that node x is a **lesser descendant** of y if and only if $x \in T(y.left)$. When this is the case, we also say that y is a **greater ancestor** of x . When y is the first greatest ancestor encountered on the path from x to the root, we say that y is **deepest greater ancestor** of x . For example, in Figure 32.3, node q is the deepest lesser ancestor of x .
- We say that the node x is a **greater descendant** of y if and only if $x \in T(y.right)$. When this is the case we also say that y is a **lesser ancestor** of x . When y is the first lesser ancestor encountered on the path from x to the root, we say that y is the **deepest lesser ancestor** of x .

For an element e that is not in the collection, the first and last insert positions are the same. While a new element could be added in any insert position, we use the last insert position in the provided implementation.

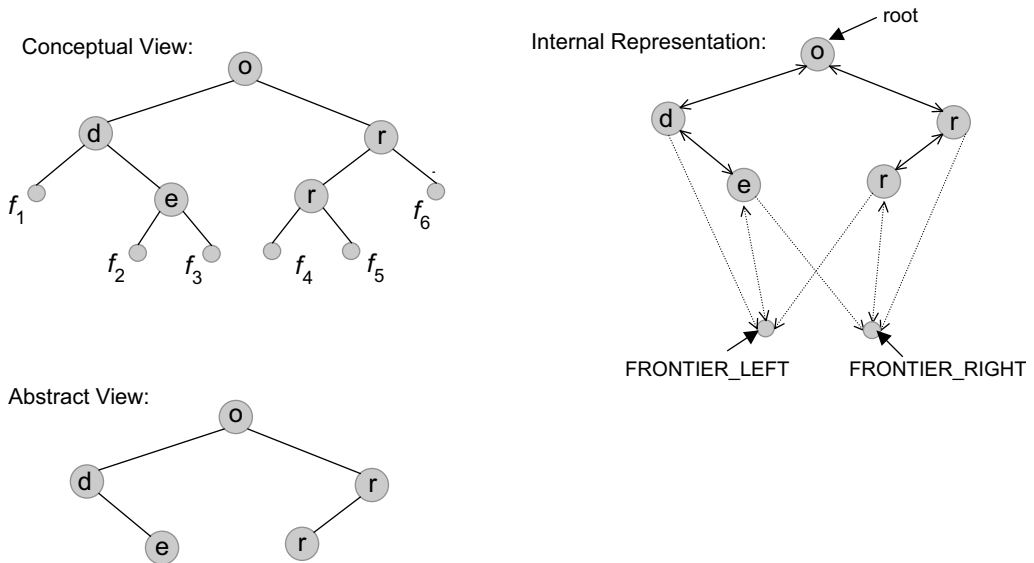
Abstraction Function: Let BST be a binary search tree instance. As for the `AbstractSearchTree`, the abstraction function is

$$AF(BST) = seq(root)$$

where seq is defined based on an inorder traversal of the tree.

Design Notes: Conceptually, a tree of n elements has $n + 1$ frontier nodes. Figure 32.2 shows a binary search tree in three different views: the conceptual view with the $n + 1$ frontier nodes, the internal representation in which arrows going downward are child pointers and arrows going upward are parent pointers, and the abstract view that we use in the figures, in which the frontier nodes are not shown.

Having the internal representation include $n + 1$ frontier nodes would greatly increase the space usage. One possible solution would be to have a single frontier node shared throughout the binary search tree. The problem with this solution is that to find the predecessor or successor for an element not in the collection it is important to know if a frontier node was reached by way of the left or right child of its parent. A boolean variable could be added to the frontier node but that would require the frontier node to be a special subclass of the binary search tree node, creating additional code complexity.

**Figure 32.2**

Three views of a populated example for a binary search tree holding the letters of “order,” inserted in the order they appear in the word. In the internal representation, the parent reference from *FRONTIER_LEFT* references the leaf node that preceded the most recent unsuccessful search that ended at it. Likewise, the parent reference from *FRONTIER_RIGHT* references the leaf node that preceded the most recent unsuccessful search that ended at it.

Another possible solution to this problem is to have the *find* method return an object that encapsulates both a reference to the binary search tree node *x* where the search ended, and a boolean to indicate if *x* is a left or right child of its parent. However, all Java objects are allocated on the heap, so such a solution would create a garbage return value object for each internal search performed. To both minimize space usage and avoid special cases in the code, we create two singleton frontier nodes, *FRONTIER_L* for when the frontier node is a left child, and *FRONTIER_R* for when the frontier node is a right child. When external storage is used, this design choice is particularly important because it allows a reference to be identifiable as a frontier node without incurring the additional disk access to retrieve the object itself.

Optimizations: To improve readability, we include methods in the *BSTNode* class that relate to the structure of a node to its sibling, parents, and grandparents (e.g., *otherChild*, *sameSideChild*). These methods introduce a small amount of overhead. There are also some extra methods and variables included to facilitate extending this class to a splay tree and a red-black tree. In particular, both *FRONTIER_L* and *FRONTIER_R* are defined for this purpose, and could be replaced by a single frontier node in the binary search tree class if it were not going to be extended. Similarly, if the binary search tree class were not going to be extended, the *deleteAndReplaceBy* method could be eliminated, and its calls replaced by *replaceSubtreeBy*.

Also since this method extends the abstract search tree, methods are include to map between *child[0]* and the left child, between *child[1]* and the right child, and between *data[0]* and the associated data for the node.

The tracker and iterator of the binary search tree are implemented using a reference to the tree node holding the element. Thus *substituteNode* method must interchange the tree nodes instead of just replacing the data. If trackers are not required, then the *substituteNode* method could just replace the data itself and invalidate all trackers and iterators.

32.2 Representation Properties

We inherit all of the abstract search tree properties. *REACHABLE* does not change. We give an equivalent definition for *INORDER* that is specialized for a binary search tree. We modify *PARENT* and *FRONTIER* to incorporate the two frontier nodes. More specifically, *FRONTIER* ensures that the left frontier node is always used as a left child, and that the right frontier node is always used as a right child.

Since this is a tracked implementation, a method is needed to efficiently determine if a given node is in the collection. We achieve this by setting the left child pointer of a node to *null* when it is no longer in the collection. *INUSE* captures that convention. For efficient support of *advance* or *retreat* for a tracked element that is no longer in the collection, *REDIRECTCHAIN* is introduced.

INORDER: For all nodes x in the collection, we have that $T(x.left) \leq x.data \leq T(x.right)$.

FRONTIER: Frontier nodes are always on the correct side of the parent. That is, for all nodes x , $x.left \neq FRONTIER\_R$ and $x.right \neq FRONTIER\_L$.

PARENT: For all nodes x in use, either $x.left = FRONTIER\_L$ or $x.left.parent = x$, and either $x.right = FRONTIER\_R$ or $x.right.parent = x$. Finally, $root.parent = null$.

INUSE: A node x is in use if and only if $x.left \neq null$.

REDIRECTCHAIN: When a tracked element is removed then the tracker is considered to be positioned just before the successor of the element at that time. For all non-frontier nodes x that are not in use, the chain of *parent* pointers starting at x ends at the node that x is considered to be just before.

32.3 BSTNode Inner Class

TreeNode

↳ **BSTNode**

In this section, we describe the *TreeNode* inner class that supports many of the methods of the *BinarySearchTree* class. Each tree node has the following four instance variables:

protected E data;	//the data element
protected BSTNode parent;	//reference to its parent (null for root)
protected BSTNode left = FRONTIER_L;	//reference to its left child
protected BSTNode right = FRONTIER_R;	//reference to its right child

The constructor takes *data*, the element to be held in this node.

```
protected BSTNode(E data) {
    this.data = data;
}
```

The method *size* returns the number of elements held in this node. By definition, *size* is always 1 for a binary search tree.

```
final protected int size() {
    return 1;
}
```

The method *capacity* returns the maximum number of elements this node can accommodate. By definition, it is 1.

```
final protected int capacity() {
    return 1;
}
```

The *isFrontier* method returns *true* if and only if this node is a frontier node.

```
final protected boolean isFrontier() {
    return this == FRONTIER_L || this == FRONTIER_R;
}
```

The *markDeleted* method marks this node as no longer in use.

```
final void markDeleted() {
    left = null;
}
```

Correctness Highlights: It preserves INUSE. However, REDIRECTCHAIN must be preserved outside this method. ■

The *isDeleted* method returns *true* if and only if this node is not in use.

```
final boolean isDeleted() {
    return left == null;
}
```

Correctness Highlights: Follows from INUSE. ■

The AbstractCollection *data* method takes *index*, the desired index, and returns e_{index} . It throws an *IllegalArgumentException* when *index* is not 0 since a binary search tree only holds a single element in each node. It throws a *NoSuchElementException* when this node is not in use.

```
final protected E data(int index) {
    if (index != 0)
        throw new IllegalArgumentException();
    if (isDeleted())
        throw new NoSuchElementException();
    return data;
}
```

Correctness Highlights: Follows from the correctness of *isDeleted*, and the fact that each binary search tree node only includes *data[0]*. ■

The *child* method takes *index*, the index for the desired child, and returns C_{index} . It throws an *IllegalArgumentException* when *index* is not 0 or 1. Recall that C_0 is the left child, and C_1 is the right child.

```

final protected BSTNode child(int index) {
    if (index == 0)
        return left;
    else if (index == 1)
        return right;
    else
        throw new IllegalArgumentException();
}

```

Some of the binary search tree methods, such as *remove*, require moving nodes to maintain IN-ORDER. In this process, sometimes a tree node x that had been a left child is moved to become a right child. Likewise, x might moved from a right child of one node to the left child of another. While normally, nothing needs to be done when this occurs, if x is *FRONTIER_LEFT* then it must be changed to *FRONTIER_RIGHT*. To enable the calling method to update its value for x when x is a frontier node, any methods that could cause such behavior will return the replacement node for x . If x is not a frontier node then its value will be unchanged.

The *setLeft* method takes x , a reference to the node that is to become the left child of this node, and returns the possibly updated value of x .

```

final protected BSTNode setLeft(BSTNode x) {
    if (x == FRONTIER_R)                                //update x if it was FRONTIER_R
        x = FRONTIER_L;
    left = x;
    x.parent = this;
    return x;
}

```

Correctness Highlights: The conditional ensures that FRONTIER is maintained. Setting the parent pointer of x maintains PARENT, with respect to this relationship between x and this node. ■

Likewise, the *setRight* method takes x , a reference to the node that is to become the right child of this node, and returns the possibly updated value of x .

```

final protected BSTNode setRight(BSTNode x) {
    if (x == FRONTIER_L)                                //update x if it was FRONTIER_L
        x = FRONTIER_R;
    right = x;
    x.parent = this;
    return x;
}

```

Correctness Highlights: Like that for *setLeft*. ■

The *isLeftChild* method returns *true* if and only if this node is a left child. Since the root has no parent, this method requires that it is not called on the root.

```

final protected boolean isLeftChild() {
    return parent.left == this;
}

```

Correctness Highlights: From the requirements on this method, the node on which it is called has a parent. Thus, if this is not the left child of its parent, it must be the right child. ■

We provide a set of methods to access nodes with a desired relationship to this node. For the sake of efficiency, these methods are final. The *grandparent* method returns the grandparent of the node. Since the root has no parent, this method requires that it is not called on the root.

```
final protected BSTNode grandparent() {
    return parent.parent;
}
```

Correctness Highlights: By definition, the grandparent of a node is its parent's parent. ■

The method *sibling* returns the sibling of this node. Since the root has no sibling, this method requires that it is not called on the root. The return value could be a frontier node.

```
final protected BSTNode sibling() {
    return (isLeftChild()) ? parent.right : parent.left;
}
```

Correctness Highlights: By definition, the sibling of a left child is its parent's right child, and the sibling of a right child is its parent's left child. ■

A method used by the balanced search tree implementations is *sameSideChild* that returns the left child of this node when it is a left child, and the right child of this node when it is a right child. Since the root is not a left or a right child, this method requires that it is not called on the root. The return value could be a frontier node.

```
final protected BSTNode sameSideChild() {
    return (isLeftChild()) ? left : right;
}
```

Correctness Highlights: Follows from the correctness of *isLeftChild*. ■

The balanced search tree methods also make use of *otherChild* that takes *child*, one of the children of this node. It returns the other child of this node. That is, if *child* is the left child, then the right child is returned. Similarly, if *child* is the right child, then the left child is returned.

```
final BSTNode otherChild(BSTNode child) {
    return (left == child) ? right : left;
}
```

Along with the accessor methods described above, we introduce some methods that are used to set the left and right child. These methods also ensure that the frontier nodes will properly point to their parent based on the current path taken in the tree. The *replaceSubtreeBy* method takes *x*, a reference to a node, and replaces $T(\text{this})$ by $T(x)$. As with *setLeft* and *setRight*, it returns the possibly updated value of *x* since it can change when *x* is a frontier node that switches between *FRONTIER.L* and *FRONTIER.R*. This method requires that $T(\text{left}) \leq x.\text{element} \leq T(\text{right})$.

```

protected BSTNode replaceSubtreeBy(BSTNode x) {
    if (this == root) {                                     //if called on the root of the tree
        root = x;                                           //the root changes, and
        x.parent = null;                                     //x's parent must be set to null
    } else if (isLeftChild()) {                             //otherwise if a left child
        x = parent.setLeft(x);                             //reset the left child
    } else {                                                //else if a right child
        x = parent.setRight(x);                             //reset the right child
    }
    return x;
}

```

Correctness Highlights: By INORDER and the requirements placed on this method, INORDER is preserved. By the correctness of *setLeft* and *setRight*, any methods that call this method preserve PARENT and FRONTIER, with respect to x and its parent. ■

The *deleteAndReplaceBy* method also takes x , a reference to a node, and replaces $T(\text{this})$ by $T(x)$. It returns the possibly updated value of x that can change when x is a frontier nodes that switches between *FRONTIER.L* and *FRONTIER.R*. This method requires that $T(\text{left}) \leq x.\text{element} \leq T(\text{right})$. For a binary search tree, the *deleteAndReplaceBy* method simply calls *replaceSubtreeBy*. However, for the red-black tree (Chapter 34), it is overridden to enforce some additional properties. We describe the difference in semantics at that point.

```

protected BSTNode deleteAndReplaceBy(BSTNode x) {
    return replaceSubtreeBy(x);
}

```

The *substituteNode* method takes x , a reference to a node, and replaces the node on which this method is called by x . While it would be more efficient to simply update *data*, doing so would invalidate any trackers or markers that are at the current node, so a slightly more involved node substitution is performed. This method requires that $T(\text{left}) \leq x.\text{element} \leq T(\text{right})$.

```

protected void substituteNode(BSTNode x) {
    x = replaceSubtreeBy(x);
    x.setLeft(left);
    x.setRight(right);
}

```

Correctness Highlights: The call to *replaceSubtreeBy*, replaces the entire subtree $T(\text{this})$ by $T(x)$. In order to just change the node, the left and right child references for x must be updated to the left and right children of this node. By the correctness of *replaceSubtreeBy*, *setLeft*, and *setRight*, this method preserves FRONTIER and PARENT, with respect to x and its children. ■

32.4 Binary Search Tree Methods

In this section, we present the methods for the *BinarySearchTree* class.

32.4.1 Constructors and Factory Methods

The constructor creates an empty binary search tree that uses the default comparator.

```
public BinarySearchTree() {
    this(Objects.DEFAULT_COMPARATOR);
}
```

The constructor that takes *comp*, the comparator to use to order the elements, creates a binary search tree that uses the provided comparator.

```
public BinarySearchTree(Comparator<? super E> comp) {
    super(comp);
    root = FRONTIER_L;
}
```

Correctness Highlights: The abstract collection constructor initializes *comp* to the provided comparator, and initializes *size* to 0 (satisfying SIZE). All the representation properties are easily verified to hold for FORE, AFT, and the frontier nodes. REACHABLE is satisfied since an empty collection contains no nodes and nothing is reachable from *FRONTIER_L*. ■

The *createFrontierNode* factory method returns a reference to a newly created frontier node.

```
protected BSTNode createFrontierNode() {
    return createTreeNode(null);
}
```

Correctness Highlights: By default, the Java compiler initializes all references to be *null*. Only the parent is ever changed for a frontier node. This satisfies INUSE since a frontier node is never in use. ■

Similarly, the *createTreeNode* factory method takes *data*, the element to be held, and returns a reference to a new tree node holding *data*.

```
protected BSTNode createTreeNode(E data) {
    return new BSTNode(data);
}
```

32.4.2 Algorithmic Accessors

To locate element *e* in a binary search tree, first *e* is compared with the element held in the root. If they are equivalent, then the search ends. Otherwise, by INORDER, if *e* is less than the element in the root, then if *e* is in the collection it must be in the left subtree of the root. Similarly, if *e* is greater than the element in the root, then if *e* is in the collection it must be in the right subtree of the root. This process is recursively repeated on the resulting subtree until either *e* is found, or a frontier node is reached. For example, consider a search for “r” in the binary search tree shown in [Figure 32.1](#). First “r” is compared to “a” and found to be greater than “a.” The search then continues at the right subtree of “a” where “r” is compared to “b” and again found to be greater. Thus, the search again moves to the right where “r” is compared to “s” and found to be less. The search then moves to the left where “r” is found. As another example, consider a search for “e.” The path taken

will be “a” (the root), “b,” “s,” “r,” “c,” “i,” and finally ending at the left child of “i,” *FRONTIER_L*. A key observation is that the insert position for “e” is the left child of “i” which is exactly where the unsuccessful search for “e” ends.

The *find* method takes *element*, the target, and returns a reference to a node holding an equivalent element, if one exists. Otherwise it returns a reference to a frontier node at the insert position for *element* with its parent set to the node that preceded it on the search path. For example, a search for “e” in the binary search tree shown in Figure 32.1 would return *FRONTIER_L* with its parent set to the tree node that contains “i” since, as discussed above, the search path for “e” would take a path following the left child of “i.”

```
protected BSTNode find(E element) {
    BSTNode ptr = (BSTNode) root;                                //start at the root
    while (!ptr.isFrontier()) {                                    //until a frontier node is reached
        FRONTIER_L.parent = FRONTIER_R.parent = ptr;            //set frontier's parent to ptr
        int comparison = comp.compare(element, ptr.data);        //compare element to current one
        if (comparison == 0)                                     //if they are equal, return
            return ptr;
        else if (comparison < 0)                                 //if element is smaller, go left
            ptr = ptr.left;
        else                                                     //if element is larger, go right
            ptr = ptr.right;
    }
    return ptr;                                                  //not found, return ptr (frontier node)
}
```

Correctness Highlights: By REACHABLE and INORDER, this method is guaranteed to find *element* if it is held in the collection. By FRONTIER and the correctness of *isFrontier*, we are guaranteed that this method will terminate and that a comparison is never made that involves a frontier node. ■

The *findFirstInsertPosition* method takes *element*, the target. It returns a reference to the frontier node that is the first insert position for *element*. The *parent* field of the returned frontier node is set to the node that preceded it on the search path. The only difference between this method and *find*, is that it continues recursively in the *left subtree* when an internal node is equivalent to *element*. For example, a search for “r” in the binary search tree shown in Figure 32.2 would return *FRONTIER_L* with its parent set to the tree node that contains “r” (with left and right children that are frontier nodes). A search for “g” would return *FRONTIER_R* with its parent set to the tree node that contains “e.”

```
BSTNode findFirstInsertPosition(E element) {
    BSTNode ptr = (BSTNode) root;                                //start at the root
    while (!ptr.isFrontier()) {                                    //until a frontier node is reached
        FRONTIER_L.parent = FRONTIER_R.parent = ptr;            //set frontier's parent
        if (comp.compare(element, ptr.data) ≤ 0)                 //if element ≤ ptr.data, go left
            ptr = ptr.left;
        else                                                     //if element > ptr.data, go right
            ptr = ptr.right;
    }
    return ptr;                                                  //return frontier node at end of search path
}
```

Correctness Highlights: By `FRONTIER` and the correctness of `isFrontier`, we are guaranteed that this method will terminate and that a comparison is never made that involves a frontier node. By `INORDER`, the frontier node returned is the first insert position for *element*. ■

The `findLastInsertPosition` method takes *element*, the target. It returns a reference to the frontier node that is the last insert position for *element*. The *parent* field of the returned frontier node is set to the node that preceded it on the search path. The only difference between this method and `find`, is that it continues recursively in the *right subtree* when an internal node is equivalent to *element*. For example, a search for “r” in the binary search tree shown in [Figure 32.2](#) would return `FRONTIER_R` with its parent set to the tree node that contains “r” (with a left child of “r” and a right child that is `FRONTIER_R`). A search for “g” would return the same value as for `findFirstInsertPosition`.

```
protected BSTNode findLastInsertPosition(E element) {
    BSTNode ptr = (BSTNode) root;                                //start at the root
    while (!ptr.isFrontier()) {                                    //until a frontier node is reached
        FRONTIER_L.parent = FRONTIER_R.parent = ptr;            //set frontier's parent
        if (comp.compare(element, ptr.data) < 0)                //if element < ptr.data, go left
            ptr = ptr.left;
        else                                                     //if element >= ptr.data, go right
            ptr = ptr.right;
    }
    return ptr;                                                  //return frontier node at end of search path
}
```

Correctness Highlights: Like that of `findFirstInsertPosition`. ■

We now present an efficient algorithm to find the predecessor of node *x*, which exploits the structural properties of a binary search tree that follow from `INORDER`. Node *x* can be any node in $T(\text{root})$, including a node in the internal iteration order (i.e., a frontier node). This method requires that if *x* is a frontier node, then before the method is called its parent is set to the node that precedes it in the intended search path. [Figure 32.3](#) illustrates the cases that could occur. For ease of exposition, we identify each node by the element it contains, and let $T = T(x.\text{left})$. If *T* is a frontier node or $T(x.\text{left})$ is a frontier node, then *T* contains no elements. There are two cases than can occur.

Case 1: *T* is not empty. The predecessor of *x* is the rightmost node in *T*.

Case 2: *T* is empty. The predecessor of *x* is the deepest lesser ancestor of *x*. If *T* is empty and node *x* is in the left subtree for all of its ancestors, then *x* has no predecessor. If *T* is empty and node *x* has no lesser ancestor, then *x* has no predecessor.

To illustrate these cases, we consider the following examples using the binary search tree of [Figure 32.4](#). We first consider two examples where *T* is not empty.

predecessor of “v”: In this case *T* is the subtree rooted at “q.” The rightmost element in *T* is “u” which is the predecessor for “v.”

predecessor of “d”: Here *T* is the subtree holding “a,” “b,” and “c.” The rightmost element in *T* is “c” which is the predecessor for “d.”

We now consider three examples where *T* is empty.

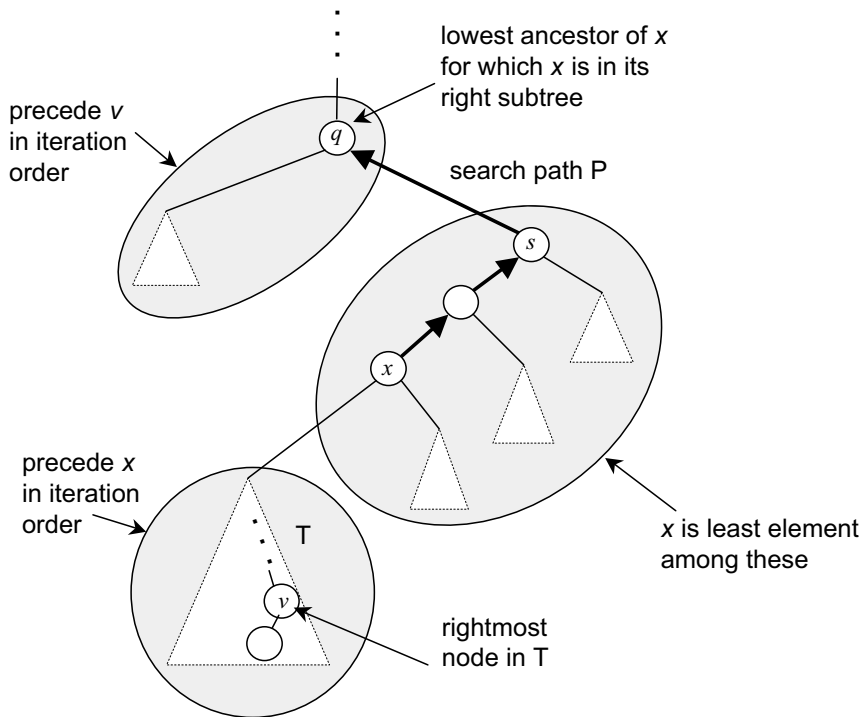


Figure 32.3

An illustration for locating the predecessor of tree node x . For simplicity, we identify each node by the element it contains. Node q is the lowest node, if any, on the search path P (which shown as a thick line with an arrow to the parent), for which x is in its right subtree.

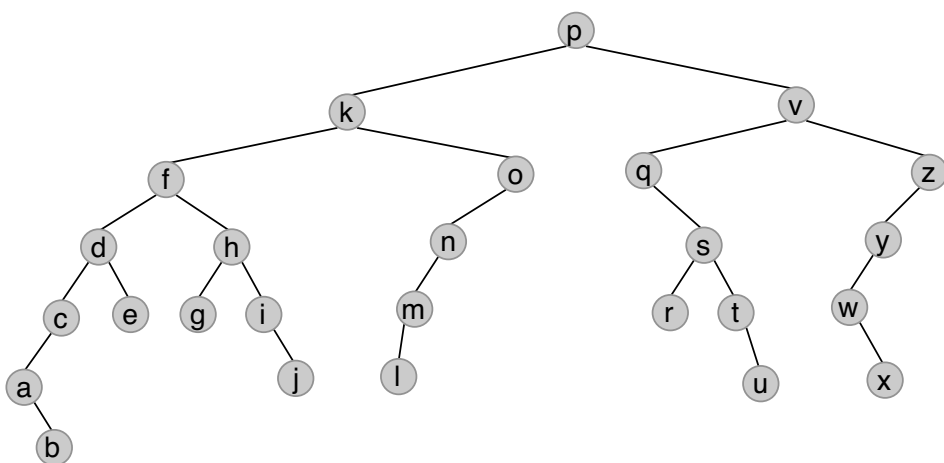


Figure 32.4

A binary search tree holding the letters in the alphabet, when inserted in a random order.

predecessor of “r”: The first node in the path from “r” to the root, for which “r” is in the right subtree, is “q.” So “q” is the deepest lesser ancestor and therefore the predecessor for “r.”

predecessor of the frontier node that is the left child of node g: The deepest lesser ancestor “f” is returned. Observe that the frontier node that is the left child of “g” would fall between “f” and “g” in the iteration order, so “f” is the correct return value.

predecessor of “a”: There is no lesser ancestor of “a,” so “a” has no predecessor.

The internal method *pred* takes *x*, target, and returns the node that holds *x*’s predecessor, or *FORE* if *x* has no predecessor. This method requires that if *x* is a frontier node, then its parent is set to the node that precedes it in *x*’s search path.

Observe that *pred* computes the predecessor of *x* as described above. If *x* is not a frontier node, and its left child is not a frontier node, then *T* exists and is non-empty. When this case occurs, the rightmost element of *T* should be returned. Otherwise, *ptr* starts at *x* moving towards the root on the search path *P* until *ptr* is no longer a left child, or *ptr* is the root. When exiting the while loop, *ptr* is either the root, or at the deepest lesser ancestor of *x* (shown as *s* in the illustration of Figure 32.3). If *ptr* reaches the root, *x* is the leftmost node which has no predecessor. Otherwise, the parent of *ptr* is returned since it is the first node in the path from *x* to the root for which *x* is in its right subtree.

```

BSTNode pred(BSTNode x) {
    if (!x.isFrontier() && !x.left.isFrontier())           //check if T exists and is non-empty
        return (BSTNode) rightmost(x.left);
    BSTNode ptr = x;
    while (ptr != root && ptr.isLeftChild())
        ptr = ptr.parent;
    if (ptr == root)
        return FORE;
    else
        return ptr.parent;
}

```

Correctness Highlights: For ease of exposition, we use the variables names that occur in Figure 32.3 to refer to tree nodes that satisfy important structural properties. Recall that the iteration order is defined by an inorder traversal of the tree. We consider the following three cases:

***x* has a non-empty left subtree *T*.** Let *v* be the rightmost element in *T*. By INORDER, *v* precedes *x* and all other elements in *T* in the iteration order. What remains is to show that all other elements either precede *v* or follow *x* in the iteration order, proving that *v* is the predecessor for *x*.

Let *q* be the deepest lesser ancestor of *x*. If *q* exists then let *s* be its right child. Otherwise, let *s* be the root. By INORDER, since *x* is in the left subtree of *s*, all elements in $T(s) - T(x)$ follow *x* in the iteration order since *x* is either in their left subtree, they are in *x*’s right subtree, or they are in the right subtree of a tree node that we have argued follows *x*.

If *q* exists, then *q* precedes *v* in the iteration order since *v* is in *q*’s right subtree. All elements in *q*’s left subtree precede *q*, so by transitivity they precede *v*. Finally, we consider the ancestors of *q* (which are not shown in Figure 32.3). If *q* is a left child then all of its ancestors are after *x* in the iteration order since *x* would be in their left subtree. If *q* is a right child then all of its ancestors would be before *q* (and thus before *v*) in the iteration order. Thus if *x*’s left subtree *T* is non-empty, then the rightmost element in *T* is the predecessor of *x*.

x has no left child and is the right child of some node on its search path. Let q be the deepest lesser ancestor of x , and let s be q 's right child. We now argue that s is the predecessor of x by showing that all elements other than x and q either precede q or follow x in the iteration order. As discussed above, the elements in q 's right subtree, other than x , all follow x in the iteration order. Similarly, we have already argued that all ancestors of q either precede q or follow x in the iteration order. The only elements that remain are those in q 's left subtree which precede q .

x has no left child and is the left child of all nodes on its search path. In this case, x is the leftmost node, which by INORDER is first in the iteration order. Thus in this case, *FORE* is the correct return value.

Termination is guaranteed by PARENT, which guarantees that when the root is reached that the parent will be *null*. This completes the correctness argument for *pred*. ■

The public *predecessor* method takes *target*, the element to search for, and returns the maximum element in this collection that is less than *target*. The target need not be in the collection. For example, in the binary search tree of Figure 32.1, the predecessor of "f" is "c." It throws a *NoSuchElementException* when no element in the collection is less than *target*.

```
public E predecessor(E target) {
    BSTNode ptr = pred(findFirstInsertPosition(target));
    if (ptr == FORE)
        throw new NoSuchElementException();
    else
        return ptr.data;
}
```

Correctness Highlights: By the correctness of *findFirstInsertPosition*, *ptr* references the frontier node in the full iteration order that precedes the first occurrence of *element*. By the correctness of *pred*, if *pred(ptr)* returns *FORE*, then *element* is the smallest element in the collection and there is no predecessor. In this case a *NoSuchElementException* is properly thrown. Otherwise *pred(ptr)* references the internal node holding the largest element in the collection that is less than *element*, and its data is the proper return value. ■

Finding the successor of a node is symmetric to finding the predecessor. If the node has a right subtree, then the leftmost node in that subtree (i.e., the node holding the first value in the subtree) is the successor. Otherwise, the deepest greater ancestor of x . For example, consider finding the successor of node "j." For nodes "i," "h," and "f," node "j" is in their left subtree. So node "h" is the lowest ancestor of node "f" for which "f" is in its left subtree. If there is no such node then x holds the last element in the collection, so *AFT* is returned.

The internal method *succ* takes x , the target, and returns the node that holds x 's successor if one exists, and *AFT* otherwise. This method requires that if x is a frontier node, then its parent is set to the node that precedes it in x 's search path.

```
BSTNode succ(BSTNode x) {
    if (!x.isFrontier() && !x.right.isFrontier())           //check if T exists and is non-empty
        return (BSTNode) leftmost(x.right);
    BSTNode ptr = x;
    while (ptr != root && !ptr.isLeftChild())
        ptr = ptr.parent;
    if (ptr == root)
```

```

        return AFT;
    else
        return ptr.parent;
}

```

Correctness Highlights: Symmetric to the argument for *pred*. ■

The public *successor* method takes *target*, the element to search for, and returns the smallest element in this collection that is greater than *target*. This method does not require that *target* be in the collection. It throws a *NoSuchElementException* when no element in the collection is greater than *target*.

```

public E successor(E target) {
    BSTNode ptr = succ(findLastInsertPosition(target));
    if (ptr == AFT)
        throw new NoSuchElementException();
    else
        return ptr.data;
}

```

Correctness Highlights: Symmetric to the argument for *predecessor*. ■

32.4.3 Content Mutators

Methods to Perform Insertion

As illustrated in [Figure 32.5](#), adding an element to a binary search tree is performed by searching for the frontier node where the search performed by *findLastInsertPosition* ends. The node holding the new element simply replaces the frontier node where the search ended. By using *findLastInsertPosition* equivalent elements occur in the iteration order in the order of insertion.

The internal method *insert* takes *element*, the new element, and inserts it into this collection. It returns a reference to the newly added node.

```

protected BSTNode insert(E element) {
    BSTNode t = createTreeNode(element);           //create a TreeNode t with the given element
    BSTNode ptr = findLastInsertPosition(element); //find frontier node at last insert position
    ptr.replaceSubtreeBy(t);                        //replace frontier node reached by t
    return t;
}

```

Correctness Highlights: By the correctness of *findLastInsertPosition*, *ptr* is a frontier node such that replacing it by *t* will maintain INORDER. INUSE is maintained by the fact that *createTreeNode* will set *data* to *element*. REDIRECTCHAIN holds since no nodes are removed from the collection by this method. Finally, SIZE is maintained by the AbstractSearchTree *add* method. ■

The *addTracked* method takes *element*, the new element and inserts *element* into the collection. It returns a tracker that tracks the new element.

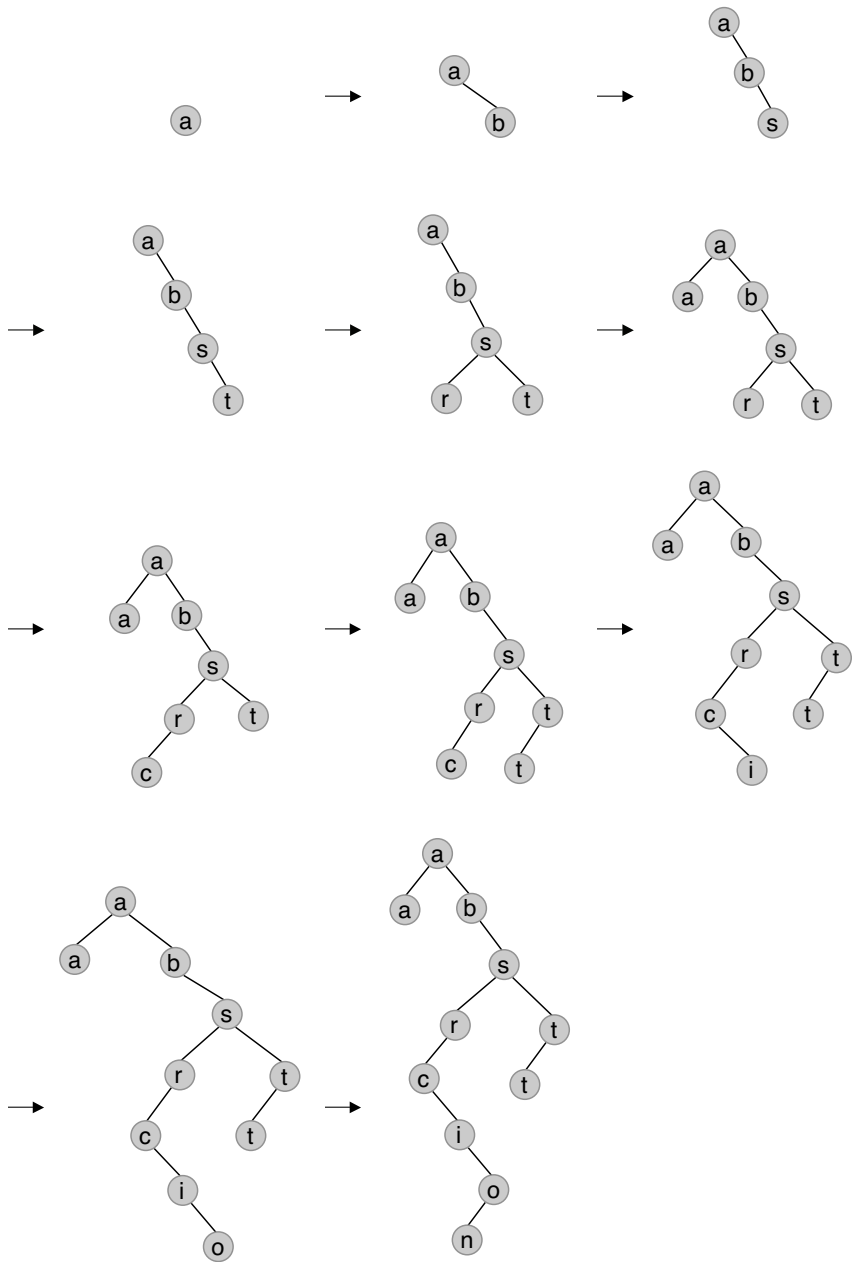
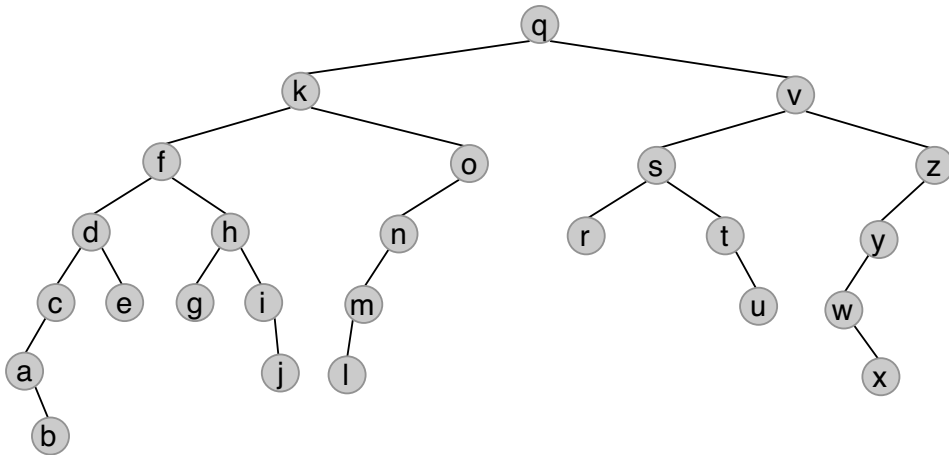


Figure 32.5
The sequence of binary search trees that result when inserting the letters of “abstraction,” in that order.

**Figure 32.6**

The binary search tree that results when “p” is removed from the binary search tree shown in [Figure 32.4](#).

```
public Locator<E> addTracked(E element) {
    size++;
    return new Tracker(insert(element));
}
```

Correctness Highlights: Follows from that of *insert*, and the fact that a tracker is created and returned. Finally, SIZE is maintained by incrementing *size*. ■

Methods to Perform Deletion

The process for removing node x is divided into three cases.

Case 1: x has no left child. In this case, the subtree rooted at x 's right child can replace x . Observe that if x also has no right child (i.e., $x.right$ is *FRONTIER_R*) then this is equivalent to just converting x to a frontier node.

Case 2: x has no right child (but does have a left child). In this case, the subtree rooted at x 's left child can replace x .

Case 3: x has a left and right child. In this case, the node y that is x 's successor (which by definition has no left child), replaces x . That is, node y is removed from the binary search tree (via Case 1 or Case 2), and then y replaces x . Alternatively, x 's predecessor could have been used. However, we use the successor since it must already be located to maintain REDIRECTCHAIN.

[Figure 32.6](#) shows the result of removing the element “p” from the binary search tree of [Figure 32.4](#). The node holding “p” in [Figure 32.4](#) has a non-empty left and right subtree, so Case 3 occurs. Thus node “p” is replaced by its successor, node “q,” which is then removed. To remove node “q” from its original position, its right subtree, the tree rooted at node “s,” replaces it. The internal method *remove* takes x , a reference to the node to remove. It throws a *NoSuchElementException* when node x is no longer in the collection (i.e., x is not in use).

```

protected void remove(TreeNode x) {
    BSTNode toRemove = (BSTNode) x;                                //to avoid casting each time
    if (toRemove.isDeleted())                                         //x already removed
        throw new NoSuchElementException();
    BSTNode successor = (BSTNode) succ(toRemove);                   //needed to update trackers
    if (toRemove.left.isFrontier())                                   //Case 1
        toRemove.deleteAndReplaceBy(toRemove.right);
    else if (toRemove.right.isFrontier())                             //Case 2
        toRemove.deleteAndReplaceBy(toRemove.left);
    else {                                                            //Case 3
        successor.deleteAndReplaceBy(successor.right);              //remove replacement
        toRemove.substituteNode(successor);
    }
    toRemove.parent = successor;                                     //preserved RedirectChain
    toRemove.markDeleted();                                         //preserves InUse
    size--;                                                         //preserves Size
}

```

Correctness Highlights: We first argue that INORDER is maintained. We individually, consider the three cases.

Case 1: x has no left child Let node y be x 's right child. Since INORDER held before this method was called, and $T(y)$ is not changed, it follows that INORDER is maintained within it. Similarly, by INORDER it follows that all elements in the subtree $T(x)$ satisfy INORDER with respect to x 's parent and the rest of the tree. Thus replacing $T(x)$ by $T(y)$ maintains INORDER.

Case 2: x has no right child Symmetric to Case 1.

Case 3: x has a left and right child By definition of the successor, if the node holding the successor for x replaces node x , INORDER is maintained since there are no elements in the collection between that held by x and its successor. Finally, the removal of the successor from its prior position in the tree preserves INORDER since it applies Case 1.

We now argue that the other representation properties are maintained. By construction REACHABLE is maintained since all nodes, except x , that had been reachable are still reachable. The *deleteAndReplaceBy* and *substituteNode* methods maintain FRONTIER and PARENT for the parent and child references that change. The update of $x.parent$ preserves REDIRECTCHAIN, and the correctness of *succ* ensures that *successor* is the successor for x . Similarly, by the correctness of *markDeleted*, marking x as deleted preserves INUSE. Finally, SIZE is preserved by decrementing *size*. ■

For the sake of efficiency, we use an inorder traversal to remove all nodes in the collection. Since all elements are going to be removed, for each node only two changes are required: set the parent pointer to FORE, and mark the node as deleted.

```

public void clearNodes(BSTNode x){
    if (x != null && !x.isFrontier()){
        clearNodes(x.left);
        clearNodes(x.right);
    }
}

```

```

        x.parent = AFT;
        x.markDeleted();
    }
}

```

//done with left child so can reset

Correctness Highlights: All elements are visited by the inorder traversal, so each node will have *parent* set to AFT and be marked as deleted. Thus INUSE and REDIRECTCHAIN will be satisfied after *clearNodes* is called on the root. ■

The *clear* method removes all elements from the collection.

```

public void clear(){
    clearNodes((BSTNode) root);
    root = FRONTIER_L;
    size = 0;
}

```

Correctness Highlights: INUSE and REDIRECTCHAIN are preserved by *clearNodes*. The last two lines preserve REACHABLE and SIZE. The remaining properties hold vacuously since there are no nodes in use once *clear* has completed. ■

32.4.4 Locator Initializers

The *iterator* method creates a new tracker at FORE.

```

public Locator<E> iterator() {
    return new Tracker((BSTNode) FORE);
}

```

The *iteratorAtEnd* method creates a new tracker that is at AFT.

```

public Locator<E> iteratorAtEnd() {
    return new Tracker((BSTNode) AFT);
}

```

Finally, the *getLocator* method takes *x*, the target, and returns a tracker initialized to an element equivalent to *x*. It throws a *NoSuchElementException* when there is no equivalent element in the collection.

```

public Locator<E> getLocator(E x) {
    BSTNode t = find(x);
    if (t.isFrontier())
        throw new NoSuchElementException();
    return new Tracker(t);
}

```


32.5 Tracker Inner Class

`AbstractCollection<E>.AbstractLocator<E>` implements *Locator<E>*

↳ **Tracker** implements *Locator<E>*

Each binary search tree tracker has an instance variables *node* that references the node holding the tracked element.

```
BSTNode node; //reference to the tracked node
```

The constructor takes a single argument *ptr*, a reference to the node to track.

```
Tracker(BSTNode ptr) {
    this.node = ptr;
}
```

Recall that *inCollection* returns *true* if and only if the tracked element is currently in the collection.

```
public boolean inCollection() {
    return !node.isDeleted();
}
```

The *get* method returns the tracked element. It throws a *NoSuchElementException* when tracker is not at an element in the collection.

```
public E get() {
    if (!inCollection())
        throw new NoSuchElementException();
    return node.data;
}
```

Both *advance* and *retreat* must work properly when the tracked node is no longer in use. In such cases, REDIRECTCHAIN guarantees that the successor is the first node in the collection reached when following the *parent* references starting at *node*. (AFT is returned if the tracker has no successor.)

The internal *skipRemovedElements* method takes *ptr*, reference to a node, and returns a reference to the first node that is in use reached starting at *ptr* and following the *parent* references. If no node in the collection is reached, then AFT is returned. This method performs the optimization of path compression on the redirect chain by letting all of the traversed parent references refer to the returned node.

```
private BSTNode skipRemovedElements(BSTNode ptr) {
    if (!ptr.isDeleted())
        return ptr;
    if (ptr.parent.isDeleted())
        ptr.parent = skipRemovedElements(ptr.parent);
    return ptr.parent;
}
```

Correctness Highlights: If *ptr* references an element in the collection (or if *ptr* references FORE, or AFT), then *ptr* is the correct return value. Otherwise, once an element in the col-

lection is reached by following the parent pointers, by REDIRECTCHAIN, we have reached the element in the collection that follows the tracker in the iteration order. Setting *ptr.parent* to the value returned by the recursive call achieves the path compression optimization. Termination is guaranteed by REDIRECTCHAIN. ■

The *advance* method moves the tracker to the next element in the iteration order, or AFT if the last element is currently tracked. It returns *true* if and only if the tracker is at an element of the collection after the update. It throws an *AtBoundaryException* when the tracker is already at AFT when the method is called, since there is no place to advance.

```
public boolean advance() {
    checkValidity();
    if (node == AFT)
        throw new AtBoundaryException();
    else if (isEmpty())
        node = AFT;
    else if (node == FORE)
        node = (BSTNode) leftmost();
    else if (node.isDeleted())
        node = skipRemovedElements(node);
    else
        node = (BSTNode) succ(node);
    return node != AFT; //still within collection unless AFT reached
}
```

Correctness Highlights: If the tracker is currently at AFT, an *AtBoundaryException* is properly thrown. If the collection is empty then the tracker correctly moves to AFT. Otherwise, if the tracker is at FORE then it should move to the first element in the iteration order, which is the value returned by *leftmost*. If the tracker does not start at FORE or AFT, then it is necessary to consider if the tracked element has been deleted. If so, then the tracker should be moved to the next existing element in the iteration order as correctly returned by *skipRemovedElements*. Otherwise, the tracked element is that given by *succ*.

After updating *node* using either *skipRemovedElements* or *succ*, an element in the collection is tracked, provided that *node* is not AFT. Thus, the correct value is returned. ■

The *retreat* method moves the tracker to the previous element in the iteration order, or FORE if the first element is currently tracked. It returns *true* if and only if after the update, the tracker is at an element of the collection. It throws an *AtBoundaryException* when the tracker is at FORE since then there is no place to retreat. Because *skipRemovedElements* returns the next element in the iteration order for an element no longer in the collection, the *advance* method is slightly simpler. In implementing *retreat*, we must address the situation in which new elements have been added to the collection that are between the tracker location and the element returned by *skipRemovedElements*.

For example, let “*e*,” the tracked element in a collection, have predecessor “*b*” and successor “*s*.” Suppose that *e* is removed, leaving the tracker logically between *b* and *s*. Then insert “*p*,” “*q*,” and “*r*.” Although “*p*,” “*q*,” and “*r*” are not persistent elements, since they are larger than “*e*,” the only correct return value for *retreat* is “*b*.” However, the return value from *pred(skipRemovedElements(node))* would be “*r*” since it is the current predecessor of “*s*.” The *while* loop addresses this situation.

```

public boolean retreat() {
    checkValidity();
    E oldData = node.data;
    boolean wasDeleted = node.isDeleted();
    if (node == FORE)
        throw new AtBoundaryException();
    if (wasDeleted)
        node = skipRemovedElements(node);
    if (node == AFT) {
        if (isEmpty())
            node = FORE;
        else
            node = (BSTNode) rightmost();
    } else {
        node = (BSTNode) pred(node);           //otherwise move to the predecessor
    }
    if (wasDeleted && node != FORE)           //move to before any smaller elements added
        while (comp.compare(node.data, oldData) > 0)
            node = (BSTNode) pred(node);
    return node != FORE;                       //still within collection unless AFT reached.
}

```

Correctness Highlights: If *node* is at FORE then an *AtBoundaryException* is properly thrown. If the tracked element has been deleted, then by the correctness of *skipRemovedElements* the tracker will be at the successor (among the elements in the collection when the tracked node was removed). If *node* is at AFT in an empty collection then the predecessor is FORE, and otherwise the predecessor is the last element in the iteration order, which is returned by *rightmost*.

We now consider when *node* was not at AFT. If *wasDeleted* is *false*, by the correctness of *pred*, *node* is at the correct position for *retreat*. Now consider when *wasDeleted* is *true*. In this case, *node* will be at the predecessor of its current location. However, as discussed above, it could be at an element that was added which is larger than *oldData* and is thus not the correct return value. The *while* loop continues to move to the predecessor until reaching a node in the iteration order with a value less than or equal to that of *oldData*. This node is the correct node to be tracked.

If the final value of *node* as long as *node* is not FORE, an element in the collection is tracked. Thus, the correct value is returned. ■

The *hasNext* method returns *true* if there is some element in the collection after the currently tracked element.

```

public boolean hasNext() {
    checkValidity();
    if (node == FORE)
        return !isEmpty();
    if (node == AFT)
        return false;
    if (node.isDeleted())
        return skipRemovedElements(node) != AFT;
    else

```

```

    return succ(node) != AFT;
}

```

Correctness Highlights: Like that for *advance*, except that some cases are not needed since the value of *node* is not updated. ■

As discussed in Section 5.8, the *remove* method removes the tracked element and implicitly updates the tracker to be between the predecessor and the successor in the iteration order. It throws a *NoSuchElementException* when the tracker is at FORE or AFT.

```

public void remove() {
    BinarySearchTree.this.remove(node);
    updateVersion();
}

```

32.6 Performance Analysis

The asymptotic time complexities of all public methods for the `BinarySearchTree` class are shown in Table 32.7, and the asymptotic time complexities for all of the public methods of the `BinarySearchTree Tracker` class are given in Table 32.8. For a binary search tree node x , we let h_x be defined as the length of the path from the root to x . The **height** of a binary search tree is defined as the maximum over all nodes x in the tree of h_x . If the elements are inserted based on a random permutation, then the expected value of h_x for a randomly selected x is about $2 \ln n \approx 1.382 \log_2 n$. However, in the worst case the binary search tree degenerates to a sorted list, in which case $h_x = n$ for the node x at the end of the list.

The locator constructor runs in constant time, so *iterator* and *iteratorAtEnd* take constant time. Also, since the binary search tree is an elastic implementation, *ensureCapacity* and *trimToSize* need not perform any computation, so they take constant time.

The time to locate an element (or the position where it would be inserted) is $O(h)$ since *find* takes a single pass down the tree, spending constant time at each node. Thus, *contains*, *getEquivalentElement*, and *getLocator* take $O(h)$ time. Similarly, since it takes $O(1)$ to insert a node once its location is found, *add* and *addTracked* take $O(h)$ time. The *min* method takes a single pass down the tree always going to the left child, and the *max* method takes a single pass down the tree always going to the right child. Thus, both take $O(h)$ time.

The *predecessor* and *successor* methods are a little more involved. For ease of exposition, we consider the complexity of *pred* that finds the predecessor node. If the given node x is either a frontier node, or a leaf whose left child is a frontier node, then the predecessor is obtained by following the parent pointers until either the predecessor is found or the root is reached. Thus this takes $O(h)$ time. Otherwise, the predecessor is the maximum element in x 's left subtree which is found by a single pass down the tree from x which takes $O(h)$ time. The analysis for *successor* is the same.

The *remove* method must first locate the element, which as discussed above takes $O(h)$ time. If the node to remove has two children, then its predecessor must be located, which also takes $O(h)$ time. Finally, in constant time, the pointers are adjusted. Thus, the overall time complexity for *remove* is $O(h)$.

method	time complexity
<i>ensureCapacity(x)</i>	$O(1)$
<i>iterator()</i>	$O(1)$
<i>iteratorAtEnd()</i>	$O(1)$
<i>trimToSize()</i>	$O(1)$
<i>add(o),addTracked(o)</i>	$O(h)$
<i>contains(o)</i>	$O(h)$
<i>getEquivalentElement(o)</i>	$O(h)$
<i>getLocator(o)</i>	$O(h)$
<i>min()</i>	$O(h)$
<i>max()</i>	$O(h)$
<i>predecessor(o)</i>	$O(h)$
<i>remove(o)</i>	$O(h)$
<i>successor(o)</i>	$O(h)$
<i>accept(v)</i>	$O(n)$
<i>clear()</i>	$O(n)$
<i>toArray()</i>	$O(n)$
<i>toString()</i>	$O(n)$
<i>retainAll(c)</i>	$O(n(c + h))$
<i>addAll(c)</i>	$O(c \log(n + c))$

Table 32.7 Summary of the asymptotic time complexities for the public methods when using a binary search tree to implement the OrderedCollection ADT where h is the height of the tree. In the worst case $h = n$. If the elements are inserted in a random order, the expected height is approximately $1.386 \log_2 n$.

locator method	time complexity
constructor	$O(1)$
<i>get()</i>	$O(1)$
<i>advance()</i>	$O(h)$
<i>hasNext()</i>	$O(h)$
<i>next()</i>	$O(h)$
<i>remove()</i>	$O(h)$
<i>retreat()</i>	$O(h)$

Table 32.8 Summary of the amortized time complexities for the public locator methods of the binary search tree tracker class.

The *accept*, *clear*, *toArray*, and *toString* methods perform an inorder traversal. Observe that during the traversal each node is visited exactly once and constant time is spent at each node. Thus the overall cost is $O(n)$.

The *addAll* method performs $|c|$ additions into a binary search tree that has a final size of $O(n + |c|)$. Thus the overall complexity is $O(|c| \log(n + |c|))$. The *retainAll* method must perform n searches for an element in the collection c which takes $O(n|c|)$ time and then call *remove* for up to n elements. Thus the overall time complexity is $O(n(|c| + h))$.

We now analyze the time complexity of the locator methods. The methods that just access an element take $O(1)$ time. The *skipRemovedElements* method takes time proportional to the redirect chain. However, every element in the redirect chain is the result of *remove* being called, either directly or through a tracker. By charging each *remove* method call an additional constant cost, due to path compression, *skipRemovedElements* has constant amortized cost. The *advance* and *hasNext* methods are dominated by the call to *succ* which takes $O(h)$ time. Finally, the *retreat* method has cost $O(h)$, for each call to *pred*. The cost of the calls to *pred* made within the while loop can be amortized over the executions of the *insert* methods that created the need to continue to move backwards. Thus *advance*, *hasNext*, and *retreat* have amortized cost of $O(h)$. It can be shown that any sequence of k calls to *advance* or *retreat* take amortized $O(k + \log n)$ time regardless of the height of the tree.

32.7 Quick Method Reference

BinarySearchTree Public Methods

- p. 490 *BinarySearchTree*()
- p. 490 *BinarySearchTree*(Comparator<? super E> comp)
- p. 98 void **accept**(Visitor<? super E> v)
- p. 479 void **add**(E element)
- p. 100 void **addAll**(Collection<? extends E> c)
- p. 496 Locator<E> **addTracked**(E element)
- p. 100 void **clear**()
- p. 499 void **clearNodes**(BSTNode x)
- p. 97 boolean **contains**(E value)
- p. 99 void **ensureCapacity**(int capacity)
- p. 476 E **get**(int r)
- p. 96 int **getCapacity**()
- p. 96 Comparator<? super E> **getComparator**()
- p. 98 E **getEquivalentElement** (E target)
- p. 500 Locator<E> **getLocator**(E x)
- p. 96 int **getSize**()
- p. 96 boolean **isEmpty**()
- p. 500 Locator<E> **iterator**()
- p. 500 Locator<E> **iteratorAtEnd**()
- p. 478 E **max**()
- p. 477 E **min**()
- p. 495 E **predecessor**(E target)
- p. 479 boolean **remove**(E element)
- p. 100 void **retainAll**(Collection<E> c)
- p. 496 E **successor**(E target)
- p. 97 Object[] **toArray**()
- p. 97 E[] **toArray**(E[] array)

- p. 98 String **toString()**
- p. 99 void **trimToSize()**

BinarySearchTree Internal Methods

- p. 97 int **compare**(E e1, E e2)
- p. 490** BSTNode **createFrontierNode**()
- p. 490** BSTNode **createTreeNode**(E data)
- p. 97 boolean **equivalent**(E e1, E e2)
- p. 491** BSTNode **find**(E element)
- p. 491** BSTNode **findFirstInsertPosition**(E element)
- p. 492** BSTNode **findLastInsertPosition**(E element)
- p. 476 int **getLastNodeSearchIndex**()
- p. 496** BSTNode **insert**(E element)
- p. 477 TreeNode **leftmost**()
- p. 476 TreeNode **leftmost**(TreeNode x)
- p. 494** BSTNode **pred**(BSTNode x)
- p. 498** void **remove**(TreeNode x)
- p. 477 TreeNode **rightmost**()
- p. 477 TreeNode **rightmost**(TreeNode x)
- p. 495** BSTNode **succ**(BSTNode x)
- p. 99 void **traverseForVisitor**(Visitor<? super E> v)
- p. 478 void **traverseForVisitor**(Visitor<? super E> v, TreeNode x)
- p. 98 void **writeElements**(StringBuilder s)
- p. 479 void **writeElements**(StringBuilder sb)

BinarySearchTree.BSTNode Internal Methods

- p. 485** *BSTNode*(E data)
- p. 486** int **capacity**()
- p. 486** BSTNode **child**(int index)
- p. 486** E **data**(int index)
- p. 489** BSTNode **deleteAndReplaceBy**(BSTNode x)
- p. 488** BSTNode **grandparent**()
- p. 486** boolean **isDeleted**()
- p. 486** boolean **isFrontier**()
- p. 487** boolean **isLeftChild**()
- p. 486** void **markDeleted**()
- p. 488** BSTNode **otherChild**(BSTNode child)
- p. 488** BSTNode **replaceSubtreeBy**(BSTNode x)
- p. 488** BSTNode **sameSideChild**()
- p. 487** BSTNode **setLeft**(BSTNode x)
- p. 487** BSTNode **setRight**(BSTNode x)
- p. 488** BSTNode **sibling**()
- p. 485** int **size**()
- p. 489** void **substituteNode**(BSTNode x)

BinarySearchTree.Tracker Public Methods

- p. 502** boolean **advance**()
- p. 501** E **get**()
- p. 503** boolean **hasNext**()
- p. 101 void **ignoreConcurrentModifications**(boolean ignore)
- p. 101 void **ignorePriorConcurrentModifications**()
- p. 501** boolean **inCollection**()
- p. 101 E **next**()
- p. 504** void **remove**()
- p. 502** boolean **retreat**()

BinarySearchTree.Tracker Internal Methods

- p. 501** *Tracker*(BSTNode ptr)
- p. 101 void **checkValidity**()
- p. 501** BSTNode **skipRemovedElements**(BSTNode ptr)
- p. 101 void **updateVersion**()

Chapter 34

Red-Black Tree Data Structure

AbstractCollection<E> implements *Collection*<E>

└ AbstractSearchTree<E> implements *OrderedCollection*<E>

└ BinarySearchTree<E> implements *OrderedCollection*<E>, *Tracked*<E>

└ BalancedBinarySearchTree<E> implements *OrderedCollection*<E>, *Tracked*<E>

└ **RedBlackTree**<E> implements *OrderedCollection*<E>, *Tracked*<E>

Uses: Java references

Used By: Tree sort (Sections 11.4.4 and 15.5.4), TaggedRedBlackTree (Section 49.9.5), historical event collection case study (Sections 29.1 and 50.1)

Strengths: The red-black tree is a form of a balanced binary search tree that guarantees that the height of the tree is at most $2 \log_2(n + 1)$. Through partial analysis and simulations, it has been shown that a search in a red-black tree constructed from n elements, inserted in a random order, uses about $1.002 \log_2 n$ comparisons on average [136]. The red-black tree is the best general-purpose ordered collection data structure since it has guaranteed worst-case logarithmic bounds for all methods and has very fast search time. If it is important to have constant time methods to perform iteration, it could be threaded, as illustrated with the pairing heap (Chapter 27) at the expense of increased space usage and a small amount of overhead in the time complexity.

Weaknesses: Methods such as *minimum*, *maximum*, *successor*, and *predecessor*, along with Locator methods *retreat* and *advance* take logarithmic time. In contrast, these methods run in constant time for both the sorted array and skip list data structures.

Critical Mutators: none

Competing Data Structures: If the ordered collection will be filled initially and further mutations are very rare, consider ordering the data by means of a red-black tree (or other ordered collection), or sorting a positional collection, and then copying the elements into a sorted array (Chapter 30) for faster search times. Also a sorted array is a good choice if space usage is to be minimized, or if constant time access is needed for retrieving the element at rank i . If mutations are common, and *minimum*, *maximum*, *successor*, *predecessor*, *retreat*, and *advance* are frequently called, then consider instead using a skip list (Chapter 38). If it is important that recently accessed elements need to be efficiently retrieved at the expense of increasing the search time for other elements, then a splay tree (Chapter 35) should be considered. Finally, if the data structure is large enough to require secondary storage then either a B-tree (Chapter 36) or a B+-tree (Chapter 37) should be considered.

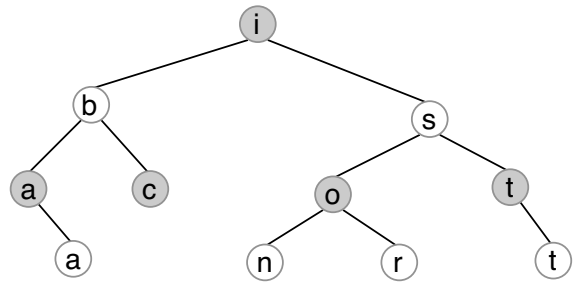


Figure 34.1
A populated example of a red-black tree for the ordered collection $\langle a, a, b, c, i, n, o, r, s, t, t \rangle$. We use the convention that a filled node is black and an unfilled node is red.

34.1 Internal Representation

A red-black tree is an extension of a binary search tree that introduces a color (red or black) for each tree node. It uses this color to determine when to perform operations that balance the search tree to ensure that it has a worst-case height of $2 \log_2(n + 1)$. All properties of the binary search tree are maintained.

Instance Variables and Constants: All of the instance variables and constants for `RedBlackTree` are inherited from `BinarySearchTree`. The `RBNode` inner class extends the `TreeNode` class, adding a boolean instance variable for the color of the node, which is either red (*true*) or black (*false*).

Populated Example: Figure 34.1 shows a populated example of a red-black tree holding the letters of “abstraction,” inserted in the order they appear in the word. Throughout this chapter we use the convention that a shaded circle represents a black node, and an unshaded circle represents a red node.

Abstraction Function: Let *RBT* be the red-black tree instance. As for the `AbstractSearchTree`, the abstraction function is

$$AF(RBT) = seq(root)$$

where *seq* is defined based on an inorder traversal of the tree. So the node color is not part of the abstraction function.

Optimizations: As for the binary search tree implementation, we have favored increased readability with some overhead for method calls to functions such as *otherChild* and *sameSideChild*. By extending `BinarySearchTree`, the differences between binary search trees and red-black trees are highlighted. However, this does slightly reduce the efficiency.

If faster iteration is needed by an application, it is possible to add a *next* and *prev* reference to each node to directly maintain the iteration order. Such a threaded implementation of a red-black tree would use a locator class like that provided for the pairing heap (Chapter 27).

34.2 Representation Properties

We inherit all of the binary search tree properties. *FRONTIER_L* and *FRONTIER_R* are both initialized to be black, and their colors never change. In addition, we add the following properties that relate to the color of the nodes.

BLACKBALANCED: The number of black nodes on any path from the root to a leaf, called the **black height**, is the same for all leaves.

NODOUBLEREDS: No red node has a red child.

ROOTBLACK: The root is always black.

Together these properties guarantee that the maximum number of nodes on any path from the root to a leaf is $2 \log_2(n + 1)$. We briefly argue how this follows from the properties. A complete binary tree with ℓ levels has $2^\ell - 1$ nodes. Thus, for a red-black tree of black height bh , $n \geq 2^{bh} - 1$. Thus $bh \leq \log_2(n + 1)$. Finally, **ROOTBLACK** and **NODOUBLEREDS** imply that in any path from a root to a leaf at least half of the nodes are black, so the number of nodes on any path from the root to a leaf is at most $2 \cdot bh \leq 2 \log_2(n + 1)$.

34.3 RBNode Inner Class

TreeNode

↑ *BSTNode*
 ↑ *RBNode*

The *RBNode* inner class adds a single instance variable to record the color of the node.

boolean colorIsRed; //true when color is red, false when color is black

The *RBNode* constructor takes *data*, the element to hold in this node, and *isRed*, the color for this node, and creates a new tree node with the specified values.

```
RBNode(E data, boolean isRed) {
    super(data);
    colorIsRed = isRed;
}
```

To improve the readability of the code, we introduce methods to check or set the color of a node. Since these methods are not overridden, they are made final to improve the efficiency.

```
final boolean isRed() { return colorIsRed; }
final boolean isBlack() { return !colorIsRed; }
final void setRed() { colorIsRed = true; }
final void setBlack() { colorIsRed = false; }
```

A red-black tree is modified through both rotations and modifying the colors. In particular, when an element is inserted, a recoloring method takes a red node with two black children, and recolors

the parent black and its children red. The *recolorRed* method checks if this node is a black node with two red children, and if so it reverses the colors of the three nodes. It returns *true* if the recoloring was performed, and *false* otherwise.

```

final boolean recolorRed() {
    if (isBlack() && ((RBNode) left).isRed() && ((RBNode) right).isRed()) {
        setRed();
        ((RBNode) left).setBlack();
        ((RBNode) right).setBlack();
        return true;
    }
    return false;
}

```

Correctness Highlights: Since the structure of the tree is not changed, all of the binary search tree properties are preserved. A key property of this method is that it preserves BLACKBALANCED. Let bh be the black height of the search tree before this method is applied, and let x be the node on which this method is called. First, if x is not black, or either of its children are not red, then no change is made. Consider when x is black and both $x.left$ and $x.right$ are black, and let p be a path from the root to a leaf that passes through x . By the definition of black height, there are bh black nodes on this path, one of them being x . After the recoloring x is now red (reducing by one the number of black nodes on the path to that point). However, the path goes through exactly one of $x.left$ or $x.right$, which has been changed from red to black (increasing by one the number of black nodes on the path). Thus, the number of black nodes on the path is unchanged. ■

Observe that deleting a black node from a red-black tree would cause a violation of BLACKBALANCED. When *deleteAndReplaceBy* is called on a black node, a method *deleteFixUp* to restore the red-black tree properties is called. Recall that *deleteAndReplaceBy* takes x , a reference to a node, and replaces $T(this)$ by $T(x)$, and returns the possibly updated value of x that can change when x is a reference to a frontier node and switches between *FRONTIER.L* and *FRONTIER.R*. It requires that $T(left) \leq x.element \leq T(right)$.

```

protected BSTNode deleteAndReplaceBy(BSTNode x) {
    x = super.deleteAndReplaceBy(x);
    if (isBlack())
        deleteFixUp((RBNode) x);
    return x;
}

```

The *substituteNode* method takes x , a reference to a node, and replaces the node on which this method is called by x . This method requires that $T(left) \leq x.element \leq T(right)$. It is just like the inherited method except that the color must also be copied.

```

protected void substituteNode(BSTNode x) {
    super.substituteNode(x);
    ((RBNode) x).colorIsRed = colorIsRed;
}

```

34.4 Methods

In this section we present the internal and public methods for the `RedBlackTree` class.

34.4.1 Constructors and Factory Methods

The constructor that takes no parameters creates an empty red-black tree that uses the default comparator.

```
public RedBlackTree() {
    this(Objects.DEFAULT_COMPARATOR);
}
```

The constructor that takes *comp*, the comparator to use to order the elements, creates a red-black tree that uses the provided comparator.

```
public RedBlackTree(Comparator<? super E> comp) {
    super(comp);
}
```

The *createFrontierNode* factory method is overridden to return a reference to a new black frontier node.

```
protected BSTNode createFrontierNode() {
    BSTNode t = new RBNode(null, false);
    return t;
}
```

The *createTreeNode* factory method takes *data*, the element to hold in the tree node. It is overridden to return a reference to a new red tree node.

```
protected BSTNode createTreeNode(E data) {
    return new RBNode(data, true);
}
```

34.4.2 Content Mutators

The only other changes required are those that maintain the red-black tree properties `BLACKBALANCED`, `NODOUBLERED`, and `ROOTBLACK`. Since `BLACKBALANCED` is a global property that would be expensive to correct if violated, modifications are made to preserve `BLACKBALANCED` at the sacrifice of `NODOUBLEREDS` and `ROOTBLACK` that can be checked and restored locally.

Methods to Perform Insertion

The basic approach to insert an element into a red-black tree is to perform a standard insertion into a binary search tree where the new node is red. Observe that `BLACKBALANCED` is trivially preserved. If the new node is the root, then it is the only element in the collection, and it can

be recolored black to preserve *RootBlack* without violating BLACKBALANCED. However, if the new node has a red parent, then NODOUBLEREDS is violated. The method *insertFixUp* takes *extraRed*, a reference to a red node that might have a red parent, and restores NODOUBLEREDS while maintaining BLACKBALANCED and ROOTBLACK. At most 2 rotations are performed by this method.

We now discuss the cases that occur for *insertFixUp*, which are illustrated in Figure 34.2. These cases are repeatedly handled until either *extraRed* has reached the root of the tree (where it can just be recolored black) or *extraRed* has a black parent in which case NODOUBLERED has been restored.

If the parent of *extraRed* is black then all three properties hold. We now consider Case 1 of Figure 34.2. Observe that by making *extraRed*'s grandparent red and its parent and uncle black, BLACKBALANCED is preserved. A possible violation of NODOUBLERED remains if *extraRed*'s great grandparent is also red. However, progress had been made since the double red violation has been moved up the tree. At this point *extraRed* is set to its grandparent since if there is a violation it is the red node with a red parent that may still need to be resolved.

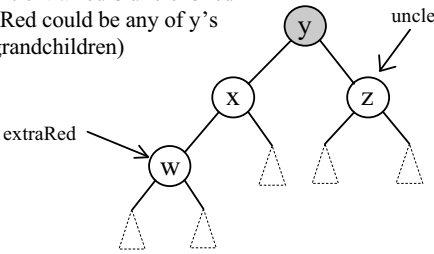
The only other possibility is that *extraRed*'s uncle is black. As illustrated in Figure 34.2, a single rotation (Case 2b), or a pair of rotations (Case 2a) along with some color changes can be performed to remove the violation of NODOUBLERED while maintaining BLACKBALANCED.

Figure 34.3 illustrates two insertions, including some intermediate steps. The top left diagram shows the state of the red-black tree after "o" is inserted (using the standard binary search tree insertion) as a left child of "r." Since "o" is a red node with a red parent, *insertFixUp* is called with the *extraRed* being node "o." Case 1 is applied since node "o" has a red uncle, which then sets the *extraRed* to node "i." Next Case 2a is applied since node "i" has a black uncle and is in a zig-zag arrangement with its parent and grandparent. Finally, Case 2a is always immediately followed by Case 2b, which completes *insertFixUp*. In the last row of Figure 34.3, "n" is inserted into the red-black tree obtained after the insertion of "o" is completed. It applies Case 2a since node "o" (the *extraRed*) has a black uncle and is in a zig-zig arrangement with its parent and grandparent. Figure 34.4 shows the full sequence of insertions for creating the red-black tree shown in Figure 34.1.

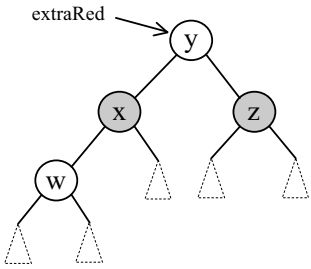
```
void insertFixUp(BSTNode extraRed) {
    while (extraRed != root && ((RBNode) extraRed.parent).isRed()) { //still have violation
        if (((RBNode) extraRed.grandparent()).recolorRed())           //Case 1
            extraRed = extraRed.grandparent();
        else {
            if (extraRed.isLeftChild() != extraRed.parent.isLeftChild()) //Case 2a, zig-zag
                extraRed = liftUp(extraRed);
            ((RBNode) extraRed.parent).setBlack();           //Case 2b, zig-zig (and rest of 2a)
            ((RBNode) extraRed.grandparent()).setRed();
            extraRed = liftUp(extraRed.parent);
        }
    }
    ((RBNode) root).setBlack(); //preserve RootBlack
}
```

Correctness Highlights: We now argue that all three properties are preserved by *insertFixUp* given that they held prior to the insertion of the new node. BLACKBALANCED is maintained throughout this method. Also, the final step of this method colors the root black, so ROOTBLACK will hold upon completion of this method. We next argue that NODOUBLERED holds when the while loop terminates. Within the while loop, we know that *extraRed* is not the root, and that both *extraRed* and its parent are red. Furthermore, we maintain the invariant that the only violation of

Case 1: extraRed's uncle is red
(extraRed could be any of y's
four grandchildren)

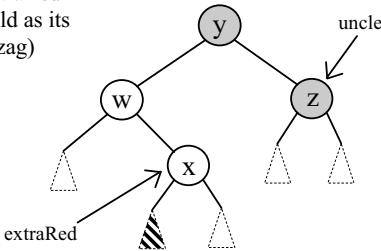


Recolor propagating extra
red up the tree

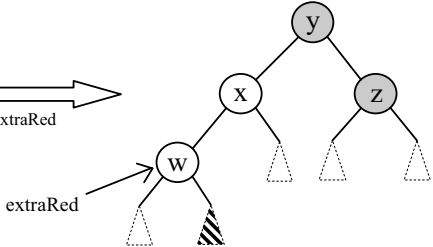


Case 2: extraRed's uncle is black
(could be mirror image)

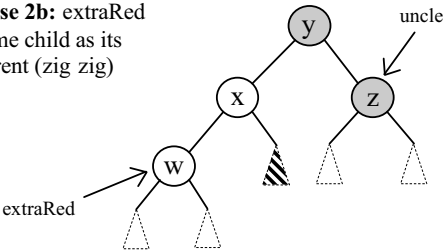
Case 2a: extraRed
opposite child as its
parent (zig zag)



Lift the extraRed



Case 2b: extraRed
same child as its
parent (zig zig)



Lift the extraRed's parent
(after which there is no
extraRed remaining)

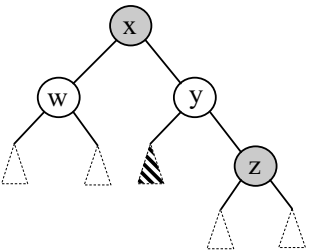


Figure 34.2

The cases for correcting the violation of **NODOUBLERED**. The node *extraRed* is the red child that might have a red parent. The node at the top of each diagram may not be the root of the tree – the portions of the tree not shown do not change. The triangles represent subtrees (possibly just a frontier node). The striped triangles are those where the parent will change.

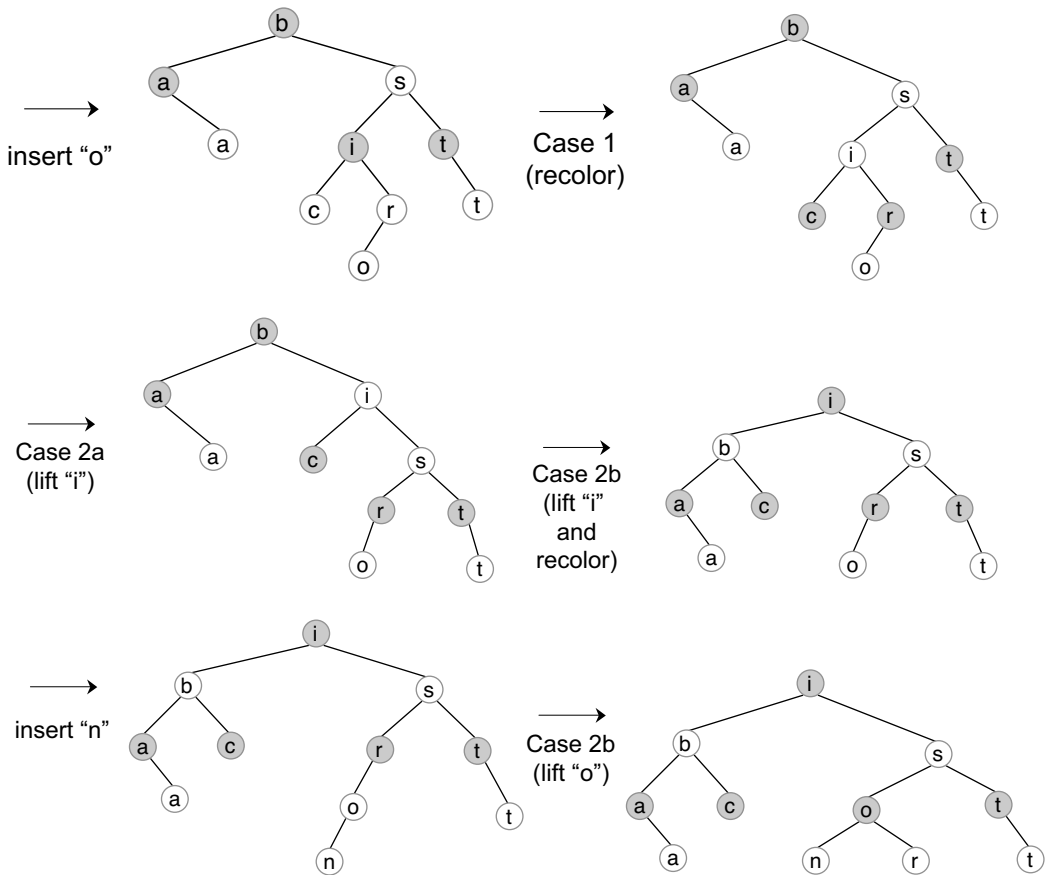


Figure 34.3
The first two row illustrates the insertion of "o." The top left diagram shows the state of the red-black tree after "o" is inserted (using the standard binary search tree insertion) as a right child of "r," and before *insertFixUp* is called. The last row illustrates the insertion of "n" into the red-black tree obtained after inserting "o."

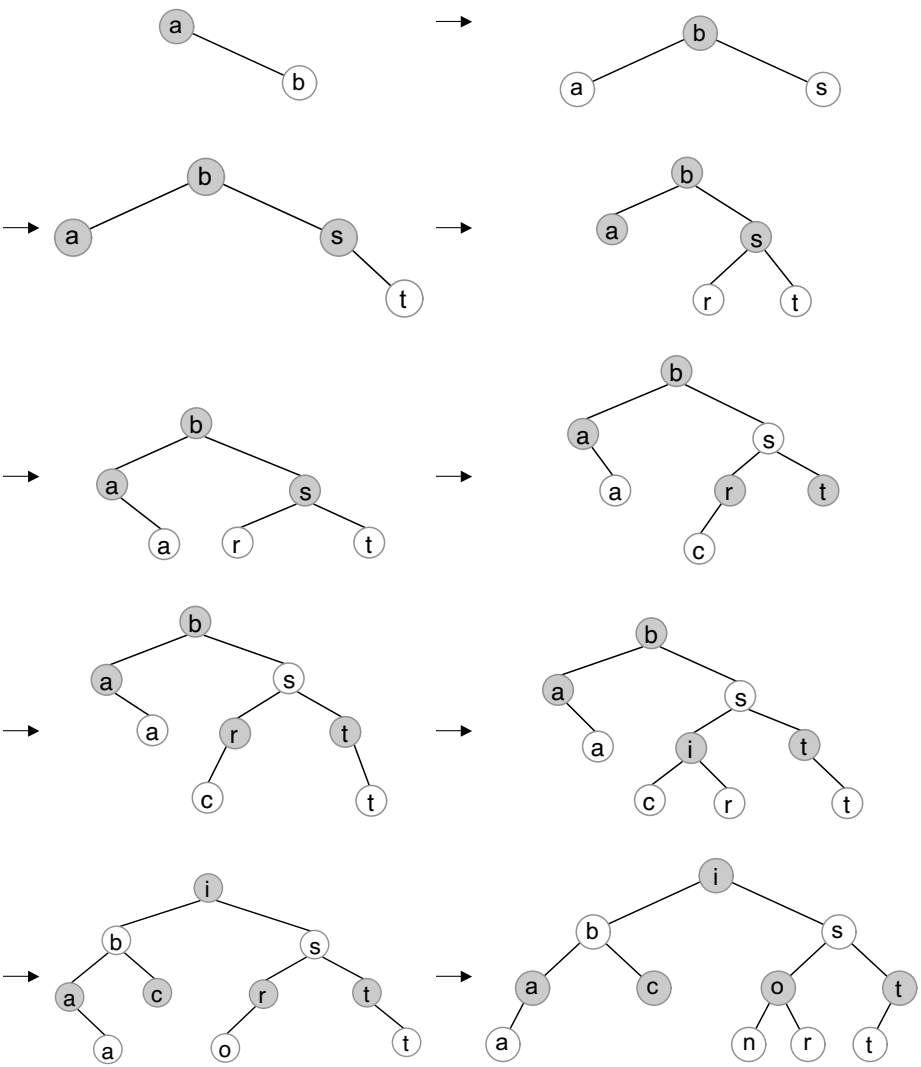


Figure 34.4

Red-black tree insertion example using the letters in “abstraction,” inserted in that order. [Figure 34.3](#) shows intermediate steps for some of these insertions.

NODOUBLERED is between *extraRed* and its parent. This invariant initially holds since a single red node was added to a red-black tree that satisfied NODOUBLERED. This invariant guarantees that the grandparent and each child, possibly a frontier node, of *extraRed* are black.

Case 1: *extraRed* has a red uncle. Since the grandparent of *extraRed* is black, the conditions for the *recolorRed* are satisfied if and only if *extraRed*'s uncle is red. In this case, *extraRed*'s parent and uncle are colored black (from red), and *extraRed*'s grandparent is colored red (from black). As discussed in the *recolorRed* method, it preserves BLACKBALANCED. At this point, there could still be a violation of NODOUBLERED between *extraRed*'s grandparent and great grandparent, but that is the only possible violation, which maintains the stated invariant. Finally, observe that this case moves *extraRed* to the position of its grandparent, bringing it closer to the root.

Case 2: *extraRed* has a black uncle. This case is further divided into two subcases.

Case 2a: *extraRed* has a zig-zag relationship with its grandparent. In this case, either *extraRed* is a right child and its parent is a left child (as shown in Figure 34.2), or *extraRed* is a left child and its parent is a right child (the mirror image of the diagram). In this case, *extraRed* is lifted by a rotation, which leads directly into Case 2b after which *insertFixUp* terminates. By the correctness of *liftUp*, BLACKBALANCED is preserved. In the diagram of Figure 34.2 it is also easy to observe that the number of black nodes on all paths in the subtree are unchanged.

Case 2b: *extraRed* has a zig-zig relationship with its grandparent. In this case, either *extraRed* is a left child and its parent is a left child (as shown in Figure 34.2), or *extraRed* is a right child and its parent is a right child (the mirror image of the diagram). In this case, the *extraRed*'s parent is lifted through a rotation. In addition, *extraRed*'s parent and grandparent exchange colors. By the correctness of *liftUp*, BLACKBALANCED is preserved. Also, observe that the color of *extraRed*'s grandparent remains black, so no new double red violation is created. Thus, NODOUBLERED is restored, and the while loop terminates.

Next we argue that the while loop will terminate. In Case 2, the while loop terminates, and whenever Case 1 is executed, *extraRed* moves closer to the root. By PARENT, eventually *extraRed* will reach the root, at which time the loop will terminate. In fact, the body of the while loop can execute at most $h/2$ times since the violation moves two steps closer to the root at each iteration.

Finally, upon termination the root is colored black. If the *root* was already black, then no properties are affected. If the *root* is red, this change restores ROOTBLACK. Changing a node to black cannot cause a violation of NODOUBLERED, so it continues to hold. Finally, changing the root from red to black preserves, BLACKBALANCED by increasing the number of blacks by 1 on every path from the root to a leaf. It is exactly by this mechanism that the black height of the tree can grow. ■

Recall that the method *insert* takes *element*, the new element, and inserts it into this collection. It returns a reference to the newly added node. The only change required is to call *insertFixUp* after using the standard binary search tree insertion.

```
protected BSTNode insert(E element) {
    RBNode node = (RBNode) super.insert(element);
    insertFixUp(node);
    return node;
}
```

Methods to Perform Deletion

We now consider the changes for the red-black tree *remove* method as compared to the binary search tree *remove* method. Tree nodes can be only red or black, but variables in the algorithm sometimes treat a particular node as having an “extra” black. For ease of exposition, we refer to either “adding one to its blackness” of a node, or “subtracting one from the blackness” of a node. If a node is black then adding one to the blackness makes it **double black**, and removing one from the blackness makes it red. Similarly, if a node is red then adding one to its blackness makes it black. We never subtract one from the blackness of a red node. As with insertion, it is important to maintain BLACKBALANCED, since it is a global property of the tree that would be expensive to correct if violated.

Recall that the standard binary search tree *remove* method always structurally removes a node with no children or one child. An internal node is removed by replacing it by its successor, which is guaranteed to have at most one child, enabling its easy removal from the tree before it is used as a replacement. For the remainder of this discussion we assume the node x being removed x has at most one child. Recall that the binary search tree *remove* method uses *deleteAndReplaceBy* to replace x by its only child, or its right frontier node if it has no children. If x is red then no violation of BLACKBALANCED, NODOUBLEREDS, or ROOTBLACK are created. However, if x is black, then when it is removed, its child y that structurally replaces x must have one added to its blackness to preserve BLACKBALANCED. (A double black node counts as two in computing the black height.) The violation that must then be addressed is that having a double black node is not a legal configuration since a node can only be black or red.

Similar to the approach used in insertion for *extraRed*, in deletion the double black node is moved up the tree until it can be locally resolved by transferring the “extra black” to a red node in such a way that DOUBLERED and BLACKBALANCED are preserved, or until it reaches the root. If the double black node reaches the root, then it can just be treated as a standard black node, which how the black height of a red-black tree is reduced.

We now describe the *deleteFixUp* method that takes *doubleBlack*, a reference to a node that is double black. It requires that the only violation of the red-black tree properties is the existence of this one double black node. The initial call to *deleteFixUp* is made from the *RBNode deleteAndReplaceBy* method when a black node y is replaced by its child x . (If y is red then *deleteFixUp* is not called.) The semantics of *doubleBlack* is that it references a node that has “blackness” one greater than its assigned color. So if *doubleBlack* is red, then it can be colored black and there will no longer be a double black node. Observe that if black node y is replaced by its red child x , then the execution of *deleteFixUp*(x) just results in x being recolored black.

The cases that occur in *deleteFixUp* are illustrated in Figure 34.5. In the code, the double black node is only identified as the node to which the variable *doubleBlack* refers. In all figures, the double black node is highlighted by an extra circle. There is never more than one double black node. In all cases, BLACKBALANCED (when counting the double black as two), and NODOUBLEREDS are preserved.

```

void deleteFixUp(RBNode doubleBlack){           //called on a node that has an “double” black
    while (doubleBlack != root && doubleBlack.isBlack()) {           //stop if at root or red node
        RBNode sibling = (RBNode) doubleBlack.sibling();
        if (sibling.isRed()){                                           //Case 1
            sibling.setBlack();
            sibling = (RBNode) liftUp(sibling);
            sibling.setRed();
            sibling = (RBNode) sibling.otherChild(doubleBlack);
        }
        if (((RBNode) sibling.left).isBlack()
            && ((RBNode) sibling.right).isBlack()){                       //Case 2a

```

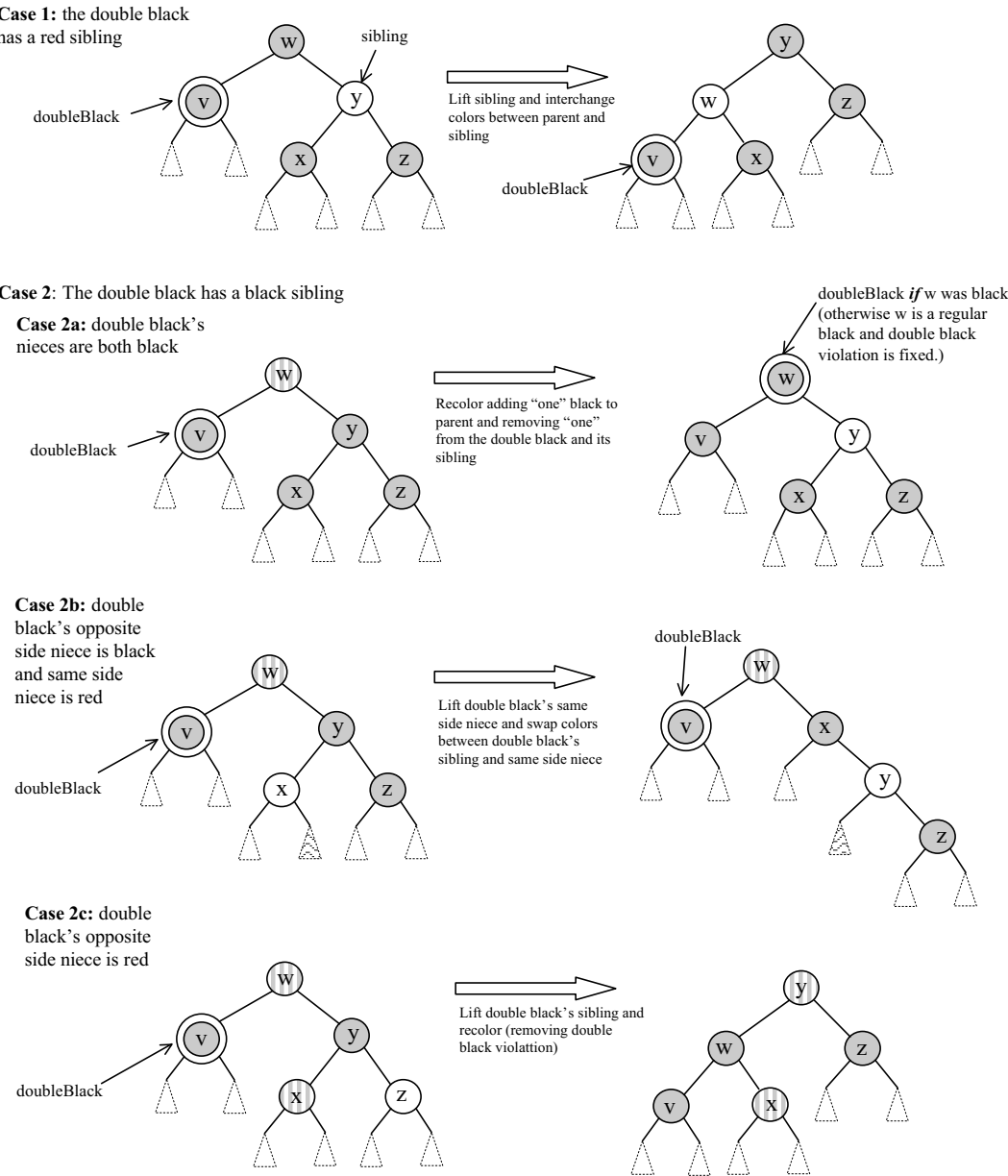
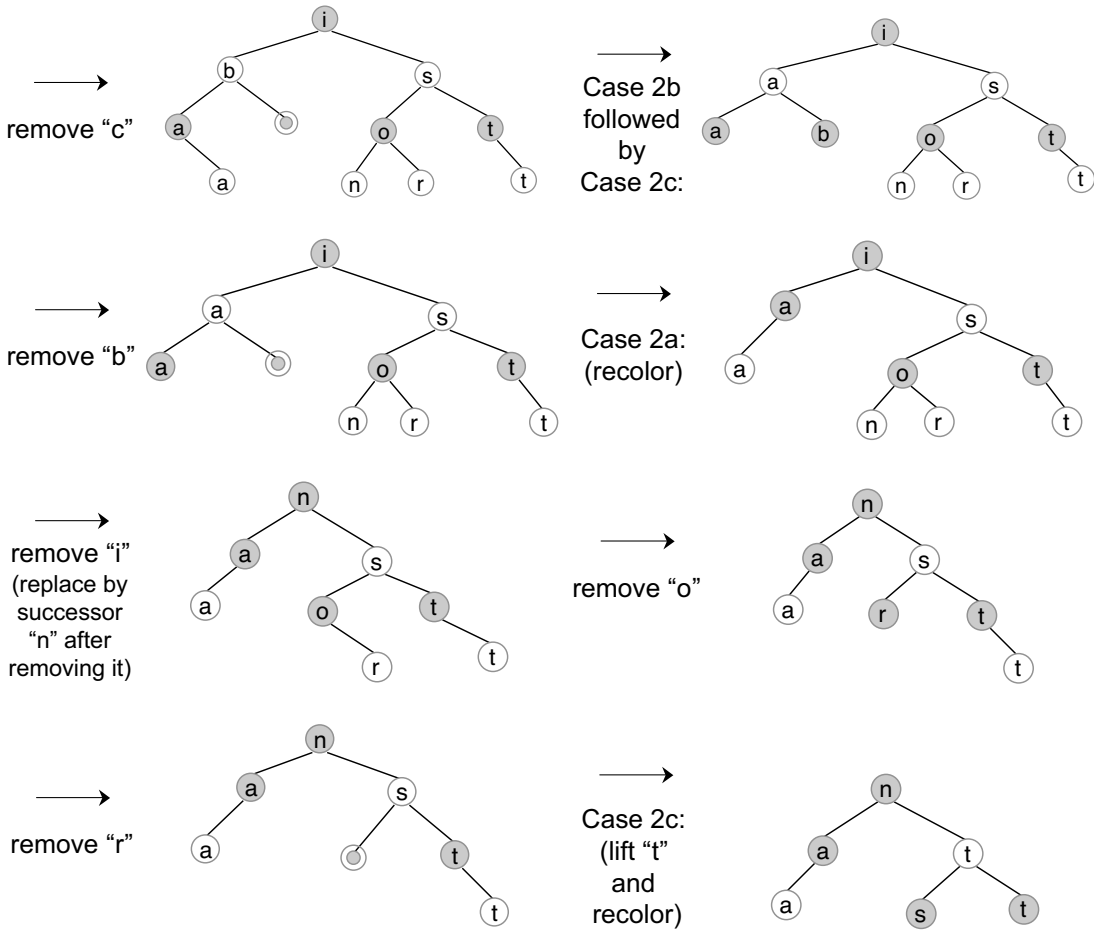


Figure 34.5

The cases for resolving the double black node, referenced by *doubleBlack*, in the main loop of *deleteFixUp*. All cases include the mirror image of what is shown. The node at the top of each diagram might not be the root of the tree. The portions of the tree not shown do not change. A black node that is shown with a letter could instead be a frontier node, including the double black. As with the insertion diagram, the triangles represent subtrees, possibly just a frontier node. The striped triangles are those for which the parent changes.



OrderedCollection

Figure 34.6
An example deletion from a red-black tree that starts with the tree shown in [Figure 34.1](#). A frontier node is shown only if it is the double black node. The double black node is shown with a circle around it.

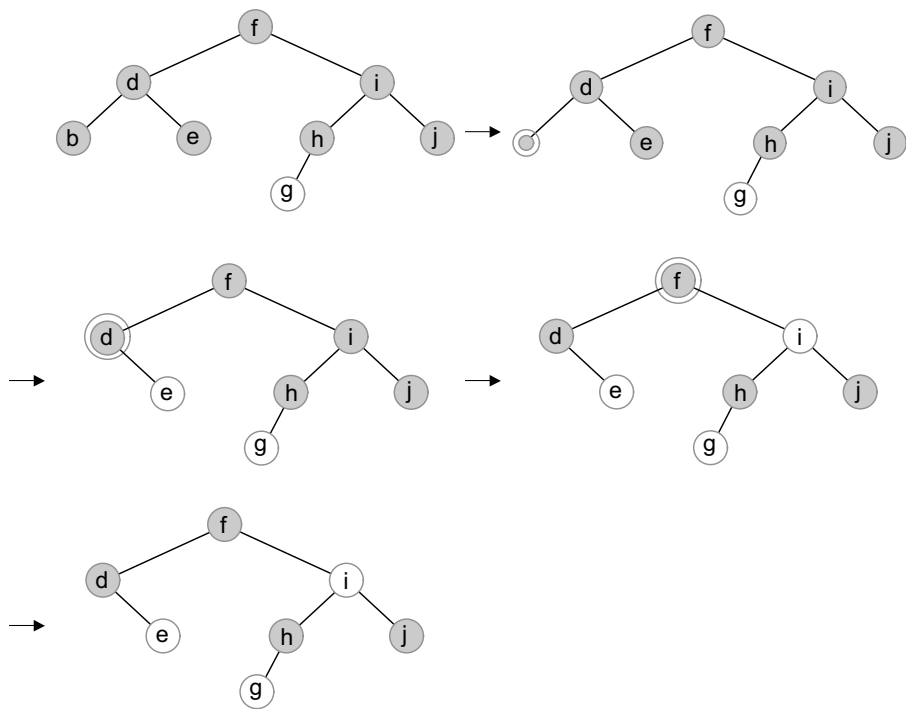


Figure 34.7
An example of deletion from a red-black tree where the recoloring percolates up the tree. In this example “b” is removed.

```

        sibling.setRed();
        doubleBlack = (RBNode) doubleBlack.parent;
    } else {
        RBNode sameSideNiece = (RBNode) sibling.sameSideChild();
        if (sameSideNiece.isBlack()) { //Case 2b (excluding color changes)
            RBNode otherSideNiece = (RBNode) sibling.otherChild(sameSideNiece);
            liftUp(otherSideNiece);
            sibling = otherSideNiece;
        }
        sibling.colorIsRed = ((RBNode) sibling.parent).colorIsRed; //Case 2c (+rest of 2b)
        ((RBNode) sibling.parent).setBlack();
        ((RBNode) sibling.sameSideChild()).setBlack();
        liftUp(sibling);
        break; //tree is legal with no doubleBlack remaining
    }
}
doubleBlack.setBlack(); //used when loop terminates with a red node as doubleBlack
}

```

Correctness Highlights: We first argue that the while loop preserves `NODOUBLEREDS` and `BLACKBALANCED`. In addition, it maintains the invariant that there is no more than one double black node. Observe that if *doubleBlack* is a black node and since double black adds one to the black height, it follows from `BLACKBALANCED` that the subtree rooted at the sibling of double black must have a black height of at least 2. Thus the sibling of double black cannot be a frontier node, which implies that double black has two nieces. Recall that a frontier node, which is black, could be either of the nieces.

Case 1: double black has a red sibling. In this case, a rotation is performed to lift the sibling so that it is now the grandparent of the double black node. The double black's parent and grandparent (previously its sibling) are then recolored. From Case 1 of Figure 34.5 it can be seen that *BlackBalanced* is preserved since the path from *y* to *v* includes three blacks both before and after the modifications. The paths through *x* and *z* both include two blacks before and after the modification. While the double black moves down the tree in this case, it will be followed by either a single execution of Case 2a at which point the double black will be eliminated, or by Case 2b or 2c, which always lead to termination.

Case 2: double black has a black sibling. This case is further divided into three subcases.

Case 2a: both nieces of double black are black. In this case `BLACKBALANCED` is preserved by adding one to the blackness of the double black's parent, and subtracting one from the blackness from both the double black node and its sibling. Since the sibling is black, we can subtract one from its blackness, making the sibling red. Also, since this case requires both nieces to be black, and one is being added to the blackness of the parent, we are guaranteed that changing the sibling to red preserves `NODOUBLEREDS`. Finally, if the double black's parent (*w* in the figure) was red, then the parent becomes black and the red-black tree is now legal. Otherwise, *w* becomes the double black node and *deleteFixUp* continues towards the root. Finally, observe that if this case is reached from Case 1, then we are guaranteed that the parent was originally red, so *deleteFixUp* will terminate.

Case 2b: opposite side niece of double black is black, but same side niece is red. In this case, a rotation and coloring changes are performed to transform Case 2b into Case

2c. It is easily verified that both `NODOUBLEREDS` and `BLACKBALANCED` are preserved. For efficiency, we do not actually make the color changes shown in Case 2b since the nodes are recolored in Case 2c, which is guaranteed to follow Case 2b. It is easy to verify that `NODOUBLEREDS` and `BLACKBALANCED` are preserved through these two steps.

Case 2c: opposite side niece of double black is red. In this case, a rotation is performed to lift double black's sibling and the colors of w and y are interchanged. After the rotation and color adjustments, both of y 's children are black and so regardless of the color of y and x , `NODOUBLEREDS` is preserved. It can be verified that the number of black nodes on the path to each subtree is the same before and after the updates, so `BLACKBALANCED` is preserved. Since all properties have been preserved and there is no double black node remaining, the red-black tree is now legal.

In summary, at each iteration, either *deleteFixUp* terminates, or the double black node is moved up towards the root. If the double black node reaches the root, then the double black can just be viewed as black, without any rotations or color changes needed. Thus, the color of the root is not changed which preserves `ROOTBLACK`. So, upon completion of *deleteFixUp*, `NODOUBLEREDS` and `BLACKBALANCED` will hold and all nodes will be either red or black.

In the analysis of the time complexity it is important to note that Case 2a, when the double black's parent is also black, is the only case in which the while loop is entered again. When this occurs, the double black is moved up the tree. Therefore, the body of the while loop can execute at most h steps. Furthermore, at most 3 rotations occur when Case 1 is followed by Case 2, which then always leads into Case 3. ■

34.5 Performance Analysis

The time complexity analysis is like that for the binary search tree, except now we have the guarantee that $h \leq 2 \log_2(n + 1) = O(\log n)$. While the method to insert and remove nodes have been modified, both *insertFixUp* and *deleteFixUp* take a single pass up the tree spending constant time at each level, so these methods have worst-case $O(\log n)$ time complexity. [Tables 34.8](#) and [34.9](#) give the resulting time complexities.

34.6 Quick Method Reference

RedBlackTree Public Methods

- p. 517 `RedBlackTree()`
- p. 517 `RedBlackTree(Comparator<? super E> comp)`
- p. 98 `void accept(Visitor<? super E> v)`
- p. 479 `void add(E element)`
- p. 100 `void addAll(Collection<? extends E> c)`
- p. 496 `Locator<E> addTracked(E element)`
- p. 100 `void clear()`
- p. 499 `void clearNodes(BSTNode x)`
- p. 97 `boolean contains(E value)`
- p. 99 `void ensureCapacity(int capacity)`
- p. 476 `E get(int r)`

method	time complexity
constructors	$O(1)$
<i>ensureCapacity(x)</i>	$O(1)$
<i>iterator()</i>	$O(1)$
<i>iteratorAtEnd()</i>	$O(1)$
<i>trimToSize()</i>	$O(1)$
<i>add(o),addTracked(o)</i>	$O(\log n)$
<i>contains(o)</i>	$O(\log n)$
<i>getEquivalentElement(o)</i>	$O(\log n)$
<i>getLocator(o)</i>	$O(\log n)$
<i>min()</i>	$O(\log n)$
<i>max()</i>	$O(\log n)$
<i>predecessor(o)</i>	$O(\log n)$
<i>remove(o)</i>	$O(\log n)$
<i>successor(o)</i>	$O(\log n)$
<i>accept(v)</i>	$O(n)$
<i>clear()</i>	$O(n)$
<i>toArray()</i>	$O(n)$
<i>toString()</i>	$O(n)$
<i>retainAll(c)</i>	$O(n c + n \log n)$
<i>addAll(c)</i>	$O(c \log(n + c))$

Table 34.8 Summary of the asymptotic time complexities for the public methods when using the red-black tree data structure to implement the OrderedCollection ADT.

OrderedCollection

locator method	time complexity
constructor	$O(1)$
<i>get()</i>	$O(1)$
<i>advance()</i>	$O(\log n)$
<i>hasNext()</i>	$O(\log n)$
<i>next()</i>	$O(\log n)$
<i>remove()</i>	$O(\log n)$
<i>retreat()</i>	$O(\log n)$

Table 34.9 Summary of the amortized time complexities for the public locator methods of the red-black tree tracker class.

- p. 96 `int getCapacity()`
- p. 96 `Comparator<? super E> getComparator()`
- p. 98 `E getEquivalentElement (E target)`
- p. 500 `Locator<E> getLocator(E x)`
- p. 96 `int getSize()`
- p. 96 `boolean isEmpty()`
- p. 500 `Locator<E> iterator()`
- p. 500 `Locator<E> iteratorAtEnd()`
- p. 478 `E max()`
- p. 477 `E min()`
- p. 495 `E predecessor(E target)`
- p. 479 `boolean remove(E element)`
- p. 100 `void retainAll(Collection<E> c)`
- p. 496 `E successor(E target)`
- p. 97 `Object[] toArray()`
- p. 97 `E[] toArray(E[] array)`
- p. 98 `String toString()`
- p. 99 `void trimToSize()`

RedBlackTree Internal Methods

- p. 97 `int compare(E e1, E e2)`
- p. 490 `BSTNode createFrontierNode()`
- p. 490 `BSTNode createTreeNode(E data)`
- p. 523 `void deleteFixUp(RBNode doubleBlack)`
- p. 97 `boolean equivalent(E e1, E e2)`
- p. 491 `BSTNode find(E element)`
- p. 491 `BSTNode findFirstInsertPosition(E element)`
- p. 492 `BSTNode findLastInsertPosition(E element)`
- p. 476 `int getLastNodeSearchIndex()`
- p. 496 `BSTNode insert(E element)`
- p. 518 `void insertFixUp(BSTNode extraRed)`
- p. 477 `TreeNode leftmost()`
- p. 476 `TreeNode leftmost(TreeNode x)`
- p. 511 `BSTNode liftUp(BSTNode y)`
- p. 494 `BSTNode pred(BSTNode x)`
- p. 498 `void remove(TreeNode x)`
- p. 477 `TreeNode rightmost()`
- p. 477 `TreeNode rightmost(TreeNode x)`
- p. 510 `void rotateLeft(BSTNode y)`
- p. 511 `void rotateRight(BSTNode y)`
- p. 495 `BSTNode succ(BSTNode x)`
- p. 99 `void traverseForVisitor(Visitor<? super E> v)`
- p. 478 `void traverseForVisitor(Visitor<? super E> v, TreeNode x)`
- p. 98 `void writeElements(StringBuilder s)`
- p. 479 `void writeElements(StringBuilder sb)`

RedBlackTree.RBNode Internal Methods

- p. 515 `RBNode(E data, boolean isRed)`
- p. 516 `BSTNode deleteAndReplaceBy(BSTNode x)`
- p. 515 `boolean isBlack()`
- p. 515 `boolean isRed()`
- p. 516 `boolean recolorRed()`
- p. 515 `void setBlack()`
- p. 515 `void setRed()`
- p. 516 `void substituteNode(BSTNode x)`

Chapter 35

Splay Tree Data Structure

AbstractCollection<E> implements *Collection*<E>

└ BinarySearchTree<E> implements *OrderedCollection*<E>, *Tracked*<E>

└ BalancedBinarySearchTree<E> implements *OrderedCollection*<E>, *Tracked*<E>

└ SplayTree<E> implements *OrderedCollection*<E>, *Tracked*<E>

Uses: Java references

Used By: TaggedSplayTree (Section 49.9.4)

Strengths: Splay trees provide excellent access time when there is high locality of reference, meaning that elements accessed recently are likely to be accessed again soon. To provide this support, splay trees use rotations to move the most recently accessed element to the root, and in the process create fairly balanced trees without keeping any additional information in the nodes. It can be shown that *insert*, *remove*, *min*, *max*, *successor*, and *predecessor* have logarithmic amortized cost. This means that any sequence of m of these operations has *worst-case* $O(m \log n)$ time complexity.

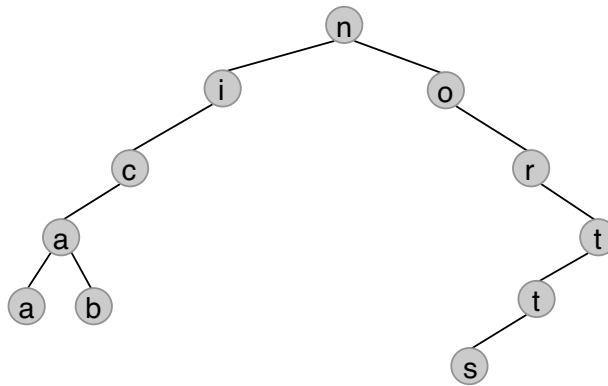
Weaknesses: In the worst-case, a splay tree can degenerate to a list.

Critical Mutators: none

Competing Data Structures: If the ordered collection will be filled initially and further mutations are very rare, consider ordering the data by means of a red-black tree (or other ordered collection), or use *addAll* to put the elements into a sorted array (Chapter 30) for faster search times. Also, a sorted array is a good choice if space usage is to be minimized, or if constant time access is needed for retrieving the element at rank i . If mutations are common, and calling *minimum*, *maximum*, *successor*, and *predecessor*, *retreat*, and *advance* is very common, then consider instead using either a red-black tree (Chapter 34) or a skip list (Chapter 38). If it is certain the elements are inserted in a random order then an unbalanced binary search tree (Chapter 32) is another option. If the data structure is large enough to require secondary storage, then either a B-tree (Chapter 36) or a B+-tree (Chapter 37) should be considered.

35.1 Internal Representation

The representation of a splay tree is identical to that of a binary search tree. In fact, the only change from a binary search tree is that whenever an element is accessed, for any reason, it is brought to the root using a sequence of rotations. All properties of the binary search tree are maintained, and no new properties are added.

**Figure 35.1**

A populated example for a splay tree holding the ordered collection obtained by inserting the letters of “abstraction,” in that order.

Populated Example: Figure 35.1 shows a populated example of a splay tree holding the letters of “abstraction,” inserted in the order they appear in the word.

Abstraction Function: Let ST be the splay tree instance. As for the `AbstractSearchTree`, the abstraction function is

$$AF(ST) = seq(root)$$

where seq is defined based on an inorder traversal of the tree.

35.2 Methods

In this section we present the internal and public methods for the `SplayTree` class.

35.2.1 Constructors

The constructor that takes no parameters creates an empty splay tree that uses the default comparator.

```

public SplayTree() {
    this(Objects.DEFAULT_COMPARATOR);
}

```

The constructor that takes *comp*, the comparator to use to order the elements, creates a red-black tree that uses the provided comparator.

```

public SplayTree(Comparator<? super E> comp) {
    super(comp);
}

```

35.2.2 Representation Mutators

The representation mutator *splay* is the only method used to restructure the tree. The *splay* method has three cases as illustrated in Figure 35.2. Case 1, called a **zig**, occurs when x is one level deeper than the desired stopping position. In this case a single call to *liftUp* is used to bring x to the desired level. In the other cases, x is brought up two levels in the tree at a time. Case 2, a **zig-zig**, occurs when x and its parent are both left children or both right children. In this case, first x 's parent is brought up one level, and then x is brought up one level. If Case 2 were removed and instead *liftUp* was called repeatedly until x reached the root, then the splay tree methods would not have $O(\log n)$ amortized complexity. Finally, Case 3, a **zig-zag**, occurs when x is a left child and its parent is a right child, or when x is a right child and its parent is a left child. In this case two consecutive calls are made to bring x up one level.

The *splay* method takes x , a reference to the node to be moved up the tree, and *stop*, a reference to the node that is the desired parent for x when this method terminates. If *stop* is *null*, then x is rotated to the root of the tree. Otherwise, the method requires that *stop* is an ancestor of x .

```
private void splay(BSTNode x, BSTNode stop) {
    while (x.parent != stop) {
        if (x.parent.parent == stop) { //Case 1 (zig)
            liftUp(x);
        }
        else if (x.parent.sameSideChild() == x) { //Case 2 (zig-zig)
            liftUp(x.parent);
            liftUp(x);
        }
        else { //Case 3 (zig-zag)
            liftUp(x);
            liftUp(x);
        }
    }
}
```

Correctness Highlights: By the correctness of *liftUp*, INORDER, and all other properties are preserved. By construction, upon termination the parent of x is *stop*. ■

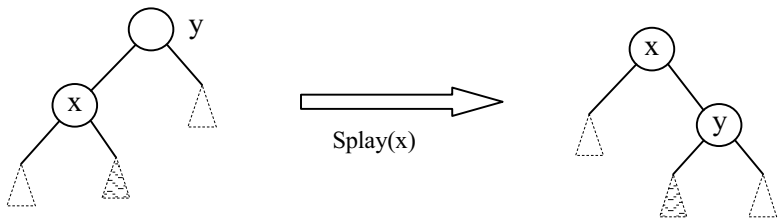
The correctness of all other methods follow from the correctness provided in the binary search tree chapter and that of *splay*, and so they are omitted in the remainder of this chapter.

35.2.3 Algorithmic Accessors

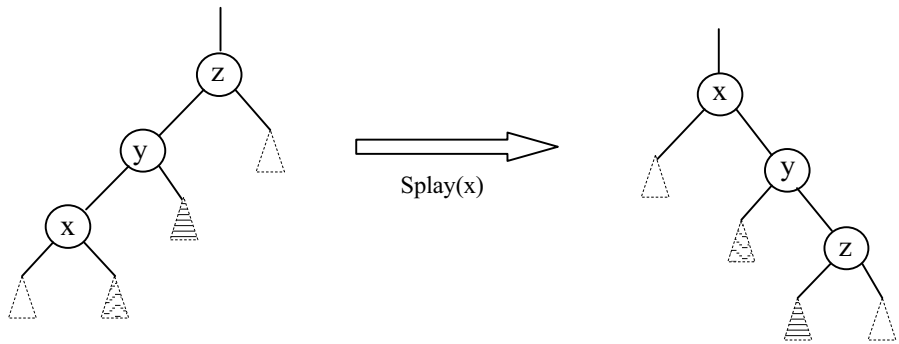
The algorithmic accessors are all overridden to call *splay* on the tree node holding the target element. The method *contains* takes *element*, the target, and returns *true* if and only if an equivalent value exists in the collection. If *element* is in the collection, it is brought to the root using the *splay* method. Otherwise, the last element accessed on the search path for *element* is brought to the root with the *splay* method.

```
public boolean contains(E element) {
    BSTNode node = find(element);
    if (!node.isFrontier()) {
        splay(node, null);
        return true;
    }
}
```

Case 1: x child of root (zig)
(could be mirror image)



Case 2: x same child as its parent (zig zig)
(could be mirror image)



Case 3: x opposite child as its parent (zig zag)
(could be mirror image)

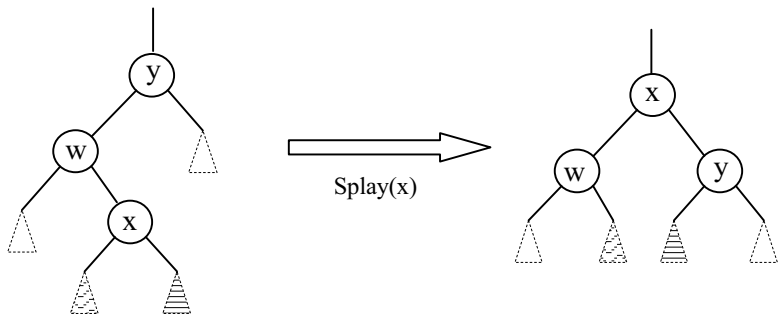


Figure 35.2
The three cases for the *splay* method. These cases are repeatedly applied until x becomes the root of the tree. The filled triangles (which could be frontier nodes) are those where the parent will change.

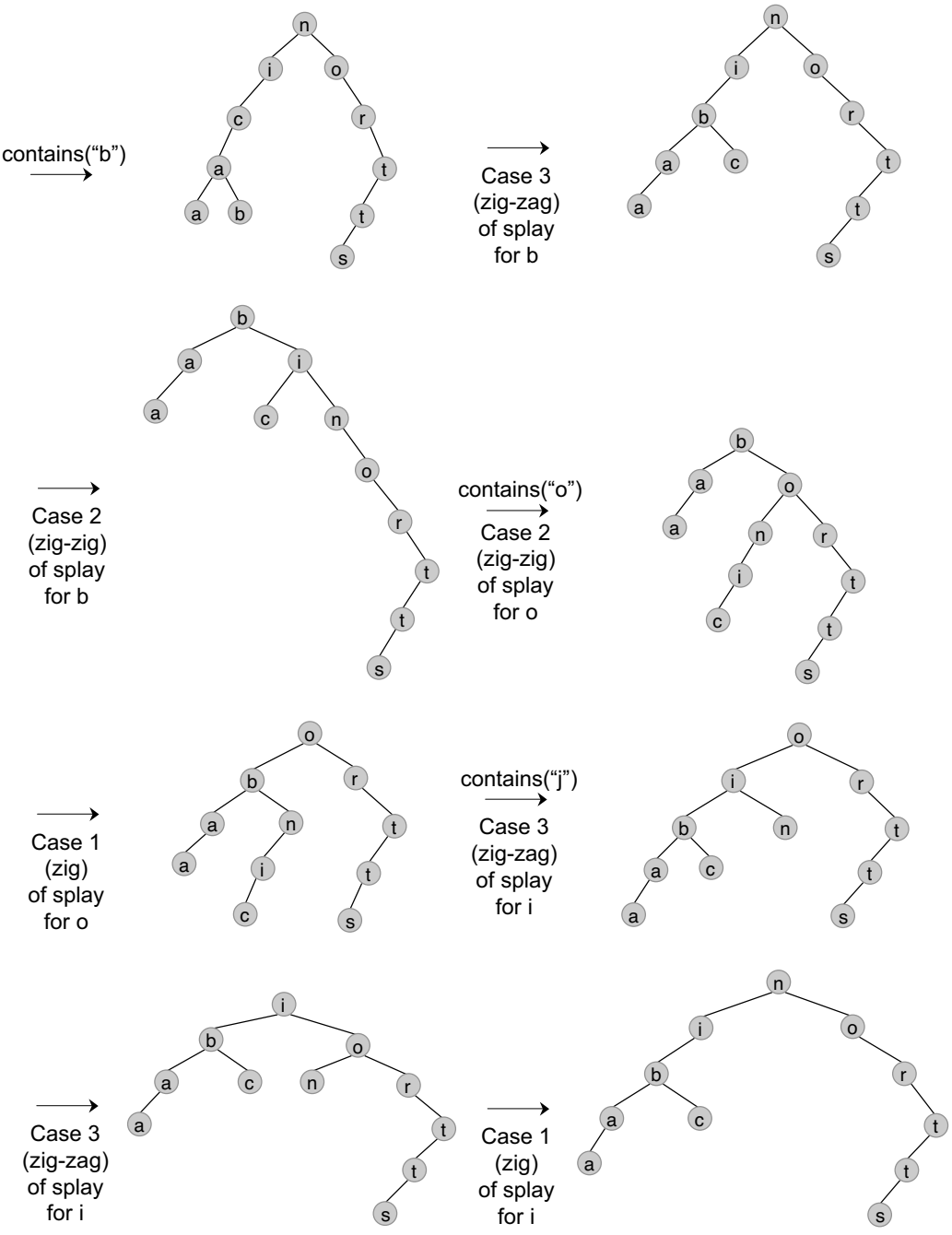


Figure 35.3
Illustration of performing the `contains` method for "b," "o," and then "j."

```

    }
    if (node != root)                                //if not found splay last node reached
        splay(node.parent, null);                    //(unless it's already the root)
    return false;
}

```

Figure 35.3 illustrates a call to *contains* for two elements in the collection, and then for an element not in the collection. First *contains* is called for “b.” After using the standard binary search tree search method to locate “b,” the *splay* method is used to bring “b” to the root. First case 3 is applied, followed by case 2. Observe that the two occurrences of “a” maintain their same relative positions in the iteration order. Next the *contains* method is called for “o” which results in applying case 2, followed by case 1. Finally, an unsuccessful search for “j” is performed. The last tree node on the search path for “j” is the one containing “i,” leading to the *splay* method being performed on “i” using case 3 followed by case 1.

The *minimum* method returns the smallest element in the collection. It throws a *NoSuchElementException* when the collection is empty. It uses *splay* to bring the minimum element to the root. Recall that since each node holds only one item, *data(0)* is that item.

```

public E min() {
    if (isEmpty())
        throw new NoSuchElementException();
    BSTNode min = (BSTNode) leftmost(root);
    splay(min, null);
    return min.data(0);
}

```

The *maximum* method returns the largest element in the collection. It throws a *NoSuchElementException* when the collection is empty. It uses *splay* to bring the maximum element to the root.

```

public E max() {
    if (isEmpty())
        throw new NoSuchElementException();
    BSTNode max = (BSTNode) rightmost(root);
    splay(max, null);
    return max.data(0);
}

```

The public *predecessor* method takes *element*, the element for which to find the predecessor. It returns the maximum value held in the collection that is less than *element*. It throws a *NoSuchElementException* when no element in the collection is less than *element*. It uses *splay* to bring the predecessor to the root.

```

public E predecessor(E element) {
    BSTNode x = findFirstInsertPosition(element);
    x = pred(x);
    if (x == FORE)
        throw new NoSuchElementException();
    else {
        splay(x, null);
        return x.data(0);
    }
}

```


The public *successor* method takes *element*, the element for which to find the successor. It returns the minimum value held in the collection that is less than *element*. It throws a *NoSuchElementException* when no element in the collection is larger than *element*. It uses *splay* to bring the successor to the root.

```
public E successor(E element) {
    BSTNode x = findLastInsertPosition(element);
    x = succ(x);
    if (x == AFT)
        throw new NoSuchElementException();
    else {
        splay(x, null);
        return x.data(0);
    }
}
```

35.2.4 Content Mutators

Methods to Perform Insertion

The *add* method takes *element*, the new element. It inserts *element* into the ordered collection and then uses *splay* to bring the new element to the root of the tree.

```
public void add(E element) {
    BSTNode node = insert(element);
    size++;
    splay(node, null);
}
```

Figure 35.4 illustrates a sequence of insertions into a splay tree. First “t” is inserted into a splay tree holding “a,” “b,” and “s,” inserted in that order. After the standard binary search tree insertion method is used to insert “t,” case 1 of the splay method brings “t” to the root. This insertion illustrates how inserting elements in sorted order into a splay tree, causes it degenerate into a sorted list. However, the time complexity for such a sequence of operations is very different from that of a binary search tree. Observe that with a binary search tree, the time for each insertion is linear in the number of elements in the collection. For the splay tree, each such insertion has constant cost. Intuitively, the fact that all the insertions are very inexpensive when building the degenerate tree allows the linear time splay method to be performed when searching for an element near the bottom of the tree. Furthermore, the fact that the splay method will restructure the tree is what ensures that each method has logarithmic amortized cost.

Next “r” is inserted into the tree. A standard binary search tree insertion is first used to insert “r” as a leaf. Then the splay method (executing case 3 followed by case 1) is used to bring “r” to the root. Figure 35.4 also illustrates the insertion of “a” and “c.” In general, any sequence of applications of case 2 and case 3, possibly followed by case 1 can occur. Figure 35.5 shows the full sequence of insertions for creating the splay tree shown in Figure 35.1.

The *addTracked* method takes *element*, the new element and inserts *element* into the collection. It returns a tracker that tracks the new element.

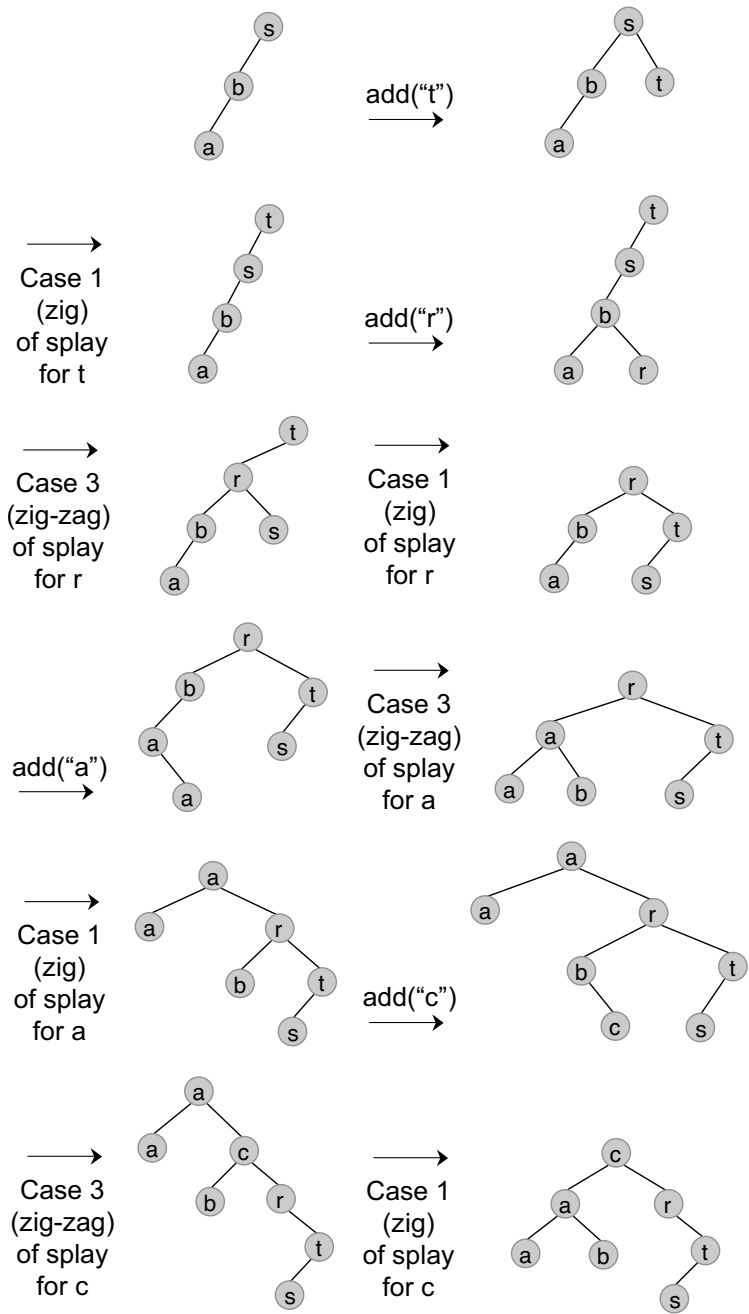


Figure 35.4
Inserting elements into a splay tree.

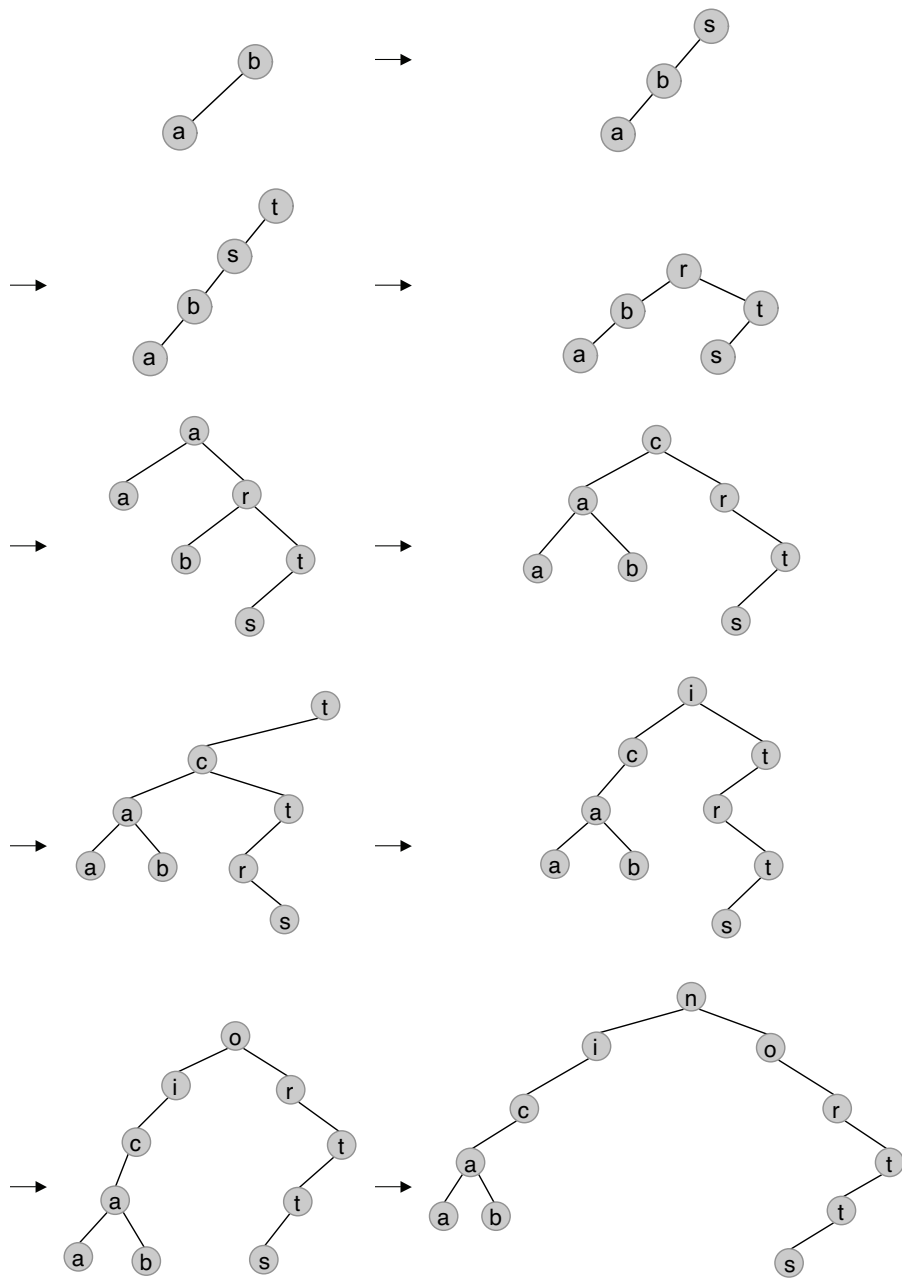
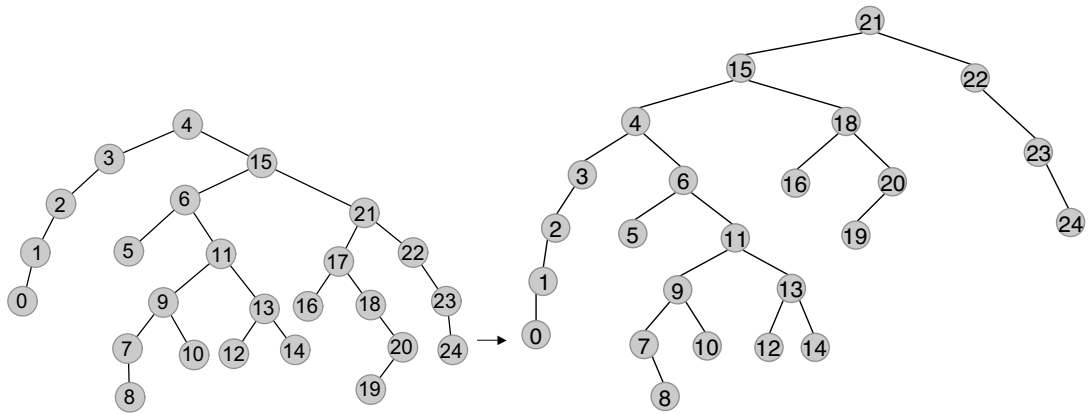


Figure 35.5
SplayTree insertion example using the letters in “abstraction,” inserted in that order. [Figure 35.4](#) shows intermediate steps from some of these insertions.

**Figure 35.6**

Removing 17, from a splay tree.

```

public Locator<E> addTracked(E element) {
    BSTNode node = insert(element);
    splay(node, null);
    size++;
    return new Tracker(node);
}

```

Methods to Perform Deletion

We override the *remove* method that takes *x*, the tree node to remove. It throws a *NoSuchElementException* when node *x* has already been removed. In order for the splay tree methods to have constant amortized cost, it is crucial that whenever a search is performed for an element that *splay* is called for that element. The intuition behind this requirement is that when there is a long search path, applying the splay operation improves the balance of the tree. (When there is a short search path the splay operation makes the tree less balanced, but it is an inexpensive operation.) The *remove* method uses *splay* to both bring the successor of *x* (if *x* has two children) to be a child of *x*, and then also uses *splay* to bring *x*'s parent to the root. The only changes between the binary search tree *remove* method, and this method are the three marked lines that call *splay*.

```

protected void remove(TreeNode x) {
    BSTNode node = (BSTNode) x;
    if (node.isDeleted())
        throw new NoSuchElementException();
    BSTNode successor = succ(node);
    if (node.left.isFrontier())
        node.deleteAndReplaceBy(node.right);
    else if (node.right.isFrontier())
        node.deleteAndReplaceBy(node.left);
    else {
        splay(successor, node);
        //splay the successor to be the child of x
    }
}

```

```

        successor.deleteAndReplaceBy(successor.right);
        node.substituteNode(successor);
    }
    if (node.parent != null)
        splay(node.parent, null);
    node.parent = successor;
    node.markDeleted();
    size--;
}

```

*//If x's parent is not already at the root
//use splay to bring x's parent to the root*

An example call to *remove* is illustrated in [Figure 35.6](#). First the node containing 18, the successor of 17, is splayed (using Case 1) to be the top of the subtree rooted at 17. This leaves 18 as the left child of 21, and 17 as the left child of 18 (not shown). Since there are no elements between 17 and its successor, it is now guaranteed to have just a single child, allowing it to be removed. Finally, the node holding 21 (the parent of 17) is brought to the root using an application of Case 2 of the *splay* method.

35.2.5 Locator Initializers

The method *getLocator* takes *x*, the element to track. It returns a locator that has been initialized at the given element. As with the *iterator* method, this method also enables navigation. It throws a *NoSuchElementException* when the given element is not in the collection. As with the other accessors, this method uses *splay* to bring *x* to the root.

```

public Locator<E> getLocator(E x) {
    BSTNode t = find(x);
    if (t.isFrontier())
        throw new NoSuchElementException();
    splay(t, null);
    return new Tracker(t);
}

```

35.3 Performance Analysis

The asymptotic time complexities of all public methods for the *SplayTree* class are shown in [Table 35.7](#), and the asymptotic time complexities for all of the public methods of the *SplayTree Tracker* class are given in [Table 35.8](#).

It can be proven [140, 146] that any sequence of *m* operations has worst case cost $O(m \log n)$. More specifically, for any sequence of *m* operations, the worst-case time is at most

$$O\left(m + \sum_{j=1}^m \log_2 n_j\right)$$

where n_j is the number of elements in the collection when the j^{th} operation in the sequence is performed. The basic property of *splay(x)* that is crucial to the analysis is that for all tree nodes on the path to the original location for *x*, their depth in the tree is roughly reduced by a factor of 2. Interestingly, if the *splay* method is replaced by one that repeatedly lifts *x* until it is at the root,

method	time complexity
constructors	$O(1)$
<i>ensureCapacity(x)</i>	$O(1)$
<i>iterator()</i>	$O(1)$
<i>iteratorAtEnd()</i>	$O(1)$
<i>trimToSize()</i>	$O(1)$
<i>add(o),addTracked(o)</i>	$O(\log n)$
<i>contains(o)</i>	$O(\log n)$
<i>getEquivalentElement(o)</i>	$O(\log n)$
<i>getLocator(o)</i>	$O(\log n)$
<i>min()</i>	$O(\log n)$
<i>max()</i>	$O(\log n)$
<i>predecessor(o)</i>	$O(\log n)$
<i>remove(o)</i>	$O(\log n)$
<i>successor(o)</i>	$O(\log n)$
<i>accept(v)</i>	$O(n)$
<i>clear()</i>	$O(n)$
<i>toArray()</i>	$O(n)$
<i>toString()</i>	$O(n)$
<i>retainAll(c)</i>	$O(n c + n \log n)$
<i>addAll(c)</i>	$O(c \log(n + c))$

Table 35.7 Summary of the amortized asymptotic time complexities for the public methods when using the splay tree data structure to implement the OrderedCollection ADT.

locator method	time complexity
constructor	$O(1)$
<i>get()</i>	$O(1)$
<i>advance()</i>	$O(\log n)$
<i>hasNext()</i>	$O(\log n)$
<i>next()</i>	$O(\log n)$
<i>remove()</i>	$O(\log n)$
<i>retreat()</i>	$O(\log n)$

Table 35.8 Summary of the amortized time complexities for the public locator methods of the splay tree tracker class.

then this property does not hold and no bound can be placed on the amortized cost of the methods. However, when using the splay method, the amortized cost of each method is $O(\log n)$. Still, no bound better than n can be given on the height of the tree because the splay tree can become a degenerate tree with a height of n . However, it has been shown that if the elements are accessed (or inserted) in a random order then the expected height is $O(\log n)$. By construction, the element accessed most recently will be at the root.

35.4 Quick Method Reference

SplayTree Public Methods

- p. 532 *SplayTree()*
- p. 532 *SplayTree*(Comparator<? super E> comp)
- p. 98 void **accept**(Visitor<? super E> v)
- p. 479 void **add**(E element)
- p. 100 void **addAll**(Collection<? extends E> c)
- p. 496 Locator<E> **addTracked**(E element)
- p. 100 void **clear**()
- p. 499 void **clearNodes**(BSTNode x)
- p. 97 boolean **contains**(E value)
- p. 99 void **ensureCapacity**(int capacity)
- p. 476 E **get**(int r)
- p. 96 int **getCapacity**()
- p. 96 Comparator<? super E> **getComparator**()
- p. 98 E **getEquivalentElement**(E target)
- p. 500 Locator<E> **getLocator**(E x)
- p. 96 int **getSize**()
- p. 96 boolean **isEmpty**()
- p. 500 Locator<E> **iterator**()
- p. 500 Locator<E> **iteratorAtEnd**()
- p. 478 E **max**()
- p. 477 E **min**()
- p. 495 E **predecessor**(E target)
- p. 479 boolean **remove**(E element)
- p. 100 void **retainAll**(Collection<E> c)
- p. 496 E **successor**(E target)
- p. 97 Object[] **toArray**()
- p. 97 E[] **toArray**(E[] array)
- p. 98 String **toString**()
- p. 99 void **trimToSize**()

SplayTree Internal Methods

- p. 97 int **compare**(E e1, E e2)
- p. 490 BSTNode **createFrontierNode**()
- p. 490 BSTNode **createTreeNode**(E data)
- p. 97 boolean **equivalent**(E e1, E e2)
- p. 491 BSTNode **find**(E element)
- p. 491 BSTNode **findFirstInsertPosition**(E element)
- p. 492 BSTNode **findLastInsertPosition**(E element)
- p. 476 int **getLastNodeSearchIndex**()
- p. 496 BSTNode **insert**(E element)
- p. 477 TreeNode **leftmost**()

- p. 476 `TreeNode` **leftmost**(`TreeNode x`)
- p. 511 `BSTNode` **liftUp**(`BSTNode y`)
- p. 494 `BSTNode` **pred**(`BSTNode x`)
- p. 498 `void` **remove**(`TreeNode x`)
- p. 477 `TreeNode` **rightmost**()
- p. 477 `TreeNode` **rightmost**(`TreeNode x`)
- p. 510 `void` **rotateLeft**(`BSTNode y`)
- p. 511 `void` **rotateRight**(`BSTNode y`)
- p. 533** `void` **splay**(`BSTNode x`, `BSTNode stop`)
- p. 495 `BSTNode` **succ**(`BSTNode x`)
- p. 99 `void` **traverseForVisitor**(`Visitor`<? super `E`> `v`)
- p. 478 `void` **traverseForVisitor**(`Visitor`<? super `E`> `v`, `TreeNode x`)
- p. 98 `void` **writeElements**(`StringBuilder s`)
- p. 479 `void` **writeElements**(`StringBuilder sb`)

Chapter 36

B-Tree Data Structure

AbstractCollection<E> implements *Collection*<E>

└ AbstractSearchTree<E> implements *OrderedCollection*<E>

└ BTree<E> implements *OrderedCollection*<E>

Uses: Java references and sorted array (Chapter 30)

Used By: TaggedBTree (Section 49.9.6)

Strengths: B-trees are designed for collections that are so large that the structure itself (not including the elements) cannot fit in main memory. When the data structure must reside in secondary storage, the dominant cost is the number of disk pages that must be accessed to locate the desired element. The B-tree is a generalization of a binary search tree designed to minimize the number of disk pages accessed by increasing the branching factor for each node. For the same n , increasing the branching factor makes the tree wider, and therefore reduces the depth of the tree. The branching factor is selected so each B-tree node fills a significant portion of the disk page, so each disk access yields as much relevant information as possible.

Weaknesses: When secondary storage is not being used, there is no time or space advantage of the increased branching factor. In fact, the time required to add or remove an element from a node is proportional to the size of the node so this cost increases with the node size. Since the cost of fetching a disk page is orders of magnitude greater than operations performed in main memory, these costs are negligible when secondary storage must be used. However, when secondary storage is not needed, a red-black tree is a better option. Also B-trees are an untracked data structure, and do not support removal of an element through a locator.

Critical Mutators: *add, clear, remove*

Competing Data Structures: If the ordered collection is held in main memory, consider a red-black tree (Chapter 34) or a skip list (Chapter 38). When secondary storage must be used and it is important to reach the successor or predecessor in constant time, the B+-tree (Chapter 37) should be considered. The B+-tree can also support very fast insertion of a batch of elements where the rest of the tree structure is added later. This is important for streaming applications (see the case study overview of Section 37.1).

When secondary storage must be used, and the size of each element is substantially larger than the portion used by the comparator, then a tagged ordered collection (Section 49.9) that wraps a B-tree is the best option. Furthermore, if the application tends to insert many elements with the same tag, then a tagged bucket ordered collection (Chapter 50) that wraps a B-tree should be considered.

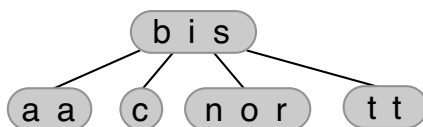


Figure 36.1

A populated example for a 2-3-4 tree (a B-tree with $t = 2$) for the ordered collection $\langle a, a, b, c, i, n, o, r, s, t, t \rangle$ obtained by inserting the elements in the order of the word “abstraction.”

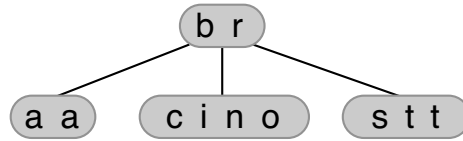
36.1 Internal Representation

A B-tree is an implementation of an abstract search tree with the additional property that all leaves are at exactly the same depth in the tree. Thus, a node with x elements has either $x + 1$ non-*null* children, or all $x + 1$ children are *null*. Each B-tree is defined by a minimum number of children t , and has the property that all nodes (except the root) hold between $t - 1$ and $2t - 1$ elements (inclusive). We refer to the number of elements held by a node as its size. The root has a size between 1 and $2t - 1$. Often, the minimum branching factor, t , is called the **order of the B-tree**.

To help understand the relationship between balanced search trees and B-trees, we discuss the close correspondence between a B-tree when $t = 2$, called a **2-3-4 tree**, and red-black trees. A red-black tree ensures that the length of the longest path from the root to a leaf in the tree is at most twice that of the shortest path from a root to a leaf. In contrast, a B-tree is completely balanced in that every path from the root to a leaf has exactly the same length, but allows the size of each node to vary by roughly a factor of 2. Figure 36.1 shows a 2-3-4 tree obtained by inserting the letters of the word “abstraction,” in that order. The red-black tree for the same sequence of insertions is shown in Figure 34.1. Any red-black tree can be converted into a 2-3-4 tree by taking each red node and combining it with its parent. Since there is never a red node with a red parent, this transformation will create nodes holding 1, 2, or 3 elements (and 2, 3, or 4 children, respectively). For example, in converting the red-black tree of Figure 34.1 to the B-tree of Figure 36.1, the nodes holding “b” and “s” are combined with the node holding the “i,” to create a node holding “bis.” The two children of “b” and two children of “s” in the red-black tree of Figure 34.1 become the four children for “bis” in the B-tree of Figure 36.1. Observe that INORDER is preserved. Two operations maintain balance in a red-black tree: recoloring and rotations. B-trees have the corresponding operations of splitting/merging nodes, and shifting elements to the left or right sibling.

While a B-tree with $t = 2$ is very similar to a red-black tree, B-trees are intended for situations in which n is so large that the data structure cannot fit in main memory (RAM). In such cases **secondary storage** (the disk) must be used. The data in secondary storage is organized in **disk pages** that are brought in as a unit. The time to access a disk page is orders of magnitude larger than the time to access a cell in main memory. For illustration, a typical access time for RAM is about 100 nanoseconds (at a cost of about \$0.08 per megabyte) whereas the access time for disk is about 10,000,000 nanoseconds (at a cost of about \$0.001 per megabyte). Therefore when secondary storage is needed, the operation costs are completely dominated by the number of disk pages that are accessed. Consequently, B-trees are designed to reduce the number of page accesses.

The order of the B-tree t is selected so that a node with $2t - 1$ elements fills a disk page. For typical page sizes, this choice allows any element, even when n is quite large, to be accessed after retrieving at most 5 disk pages. As a concrete example, suppose that a disk block holds 4096 bytes. A full node must hold:

**Figure 36.2**

A populated example for a B-tree with $t = 3$ for the ordered collection $\langle a, a, b, c, i, n, o, r, s, t, t \rangle$ obtained by inserting the elements in the order of “abstraction.”

- An array of $2t - 1$ elements,
- an array of $2t$ child references, and
- a parent reference.

In most applications, the elements held in the collection will be tagged elements (Section 49.1). Suppose that the tag is 8 bytes, and that the associated element consists of a disk block id (4 bytes) and an offset within that block (4 bytes). Then the array for the elements would require $16(2t - 1) = 32t - 16$ bytes. If each child reference and the parent reference holds a disk block id (4 bytes), then the array of child references plus the parent reference requires $4(2t + 1) = 16t + 4$ bytes. So the total space requirements for a full node would be about $48t$ bytes. Selecting t as large as possible so $48t \leq 4096$, yields a choice of $t = 85$. As we prove in Section 36.6, a B-tree of height 5, with $t = 85$ holds between 4.4 billion elements (when all nodes are minimum-sized) and 280 billion elements (when all nodes are maximum-sized).

Instance Variables and Constants: The following instance variables and constants are defined for the BTree class. Also, *root*, *size*, *comp*, *DEFAULT_CAPACITY*, and *version* are inherited from AbstractCollection. As discussed above, the integer t , is the minimum branching factor for a node (excluding the root).

```
int t = 2; //min branching factor
```

As in the binary search tree, a frontier node is created whose parent pointer can be used to retrace the last step on an unsuccessful search path.

```
final TreeNode FRONTIER = new BTreeNode();
```

Since a node can have up to $2t$ children, we introduce the global variable *curIndex* that holds the index of the child pointer used to reach the current node in the most recent search.

```
int curIndex; //index of last element located by find
```

As for the binary search tree, we introduce tree nodes for FORE and AFT.

```
TreeNode FORE = new BTreeNode();
TreeNode AFT = new BTreeNode();
```

Populated Example: Figure 36.2 shows a B-tree of order 3 holding the letters in the word “abstraction,” inserted in that order. Figure 36.1 shows the same collection for a B-tree of order 2. Finally, Figure 36.3 shows a populated example holding the letters of the alphabet, inserted in alphabetical order, in a B-tree of order 5.

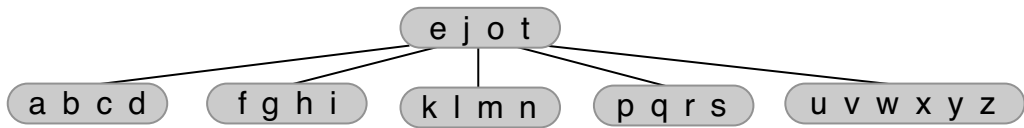


Figure 36.3

A populated example for a B-tree with $t = 5$ containing the letters of the alphabet, inserted in alphabetical order.

Terminology: We use terminology introduced for the abstract search tree (Chapter 31), and introduce the following additional terminology.

- We say that a node at its maximum size ($2t$ children) is a **full node**.
- We say that when the root has two children, or when any other node has t children, that it is a **minimum-sized** node.
- Recall that for element $x.data(i)$, we refer to $x.child(i)$ as its **left child** and $x.child(i+1)$ as its **right child**.

Abstraction Function: The abstraction function is the same as that for `AbstractSearchTree`. Namely, for the B-tree instance B , the abstraction function is

$$AF(B) = seq(root)$$

where seq is defined according to an inorder traversal of the B-tree. (See Chapter 31 for a recursive definition of seq .)

Design Notes: Recall that the binary search tree *find* method either returns a reference to a node holding the target element (if it exists), or otherwise returns a reference to a frontier node at the insert position for the target, with the frontier's parent set to the previous node on the search path. For a B-tree, a similar method is needed. However, a B-tree node can hold up to $2t - 1$ elements. One option would be for *find* to just return a reference to a node. However, then an $O(\log t)$ binary search would be necessary to recompute the desired index within that node. To avoid this unnecessary computational cost, our *find* method returns a reference to a B-tree node, and sets an index variable for the target element (if it exists) within that node. For an unsuccessful search, *find* returns a reference to *FRONTIER*, which has had its parent set to the predecessor in the search path, and sets the index variable to the insert position in *FRONTIER.parent* for the target element. Similarly both the *findFirstInsertPosition* and *findLastInsertPosition* methods both return a B-tree node, and sets an index variable for the proper insert position within the returned node.

Since Java does not support call-by-reference, we use a shared variable since we cannot return two values. In particular, we introduce a global variable *currentIndex* that is set as a side effect of one of the internal find methods. This approach uses little space. However, any methods that rely upon the value of *currentIndex* being set in another method must be synchronized. An alternative would have been to package up these two instance variables in a returned object. This alternative approach is illustrated in the Trie data structure (Chapter 41), where an object pool manages reusing these returned objects, rather than creating garbage with each call.

The provided implementation stores the B-tree nodes in the heap. If secondary storage were to be used, then methods would be needed to save and load the data to and from secondary storage. Every instance variable that currently references a B-tree node would contain a page reference and offset within that page. Decisions would also need to be made about which disk pages will remain in the heap, and when they should be rewritten back to disk.

Optimizations: Our implementation uses a bottom-up insert and remove method. If the B-tree is being stored in secondary storage, and the cache is not large enough to hold one page for each level of the tree, then a top-down insertion can avoid the need to refetch any page along the search path. We discuss the differences between these two alternatives on page 563.

A common variation of a B-tree allows each node to hold between t and $2t$ elements, as opposed to having between t and $2t$ children. In this variation, a split becomes legal only after the element is added. This variation slightly reduces the number of splits since there is more room after a split.

36.2 Representation Properties

We inherit all the AbstractSearchTree properties and add the following properties.

BALANCED: All leaf nodes are at the same height in the tree, and the frontier node is referenced only by leaf nodes. Thus, an *internal* node with e elements has $e + 1$ non-frontier children.

NODEUTILIZATION: With the exception of the root, all nodes contain at least t children where t is the order of the B-tree. The root has at least 2 children. All nodes have at most $2t$ children.

36.3 B-Tree Node Inner Class

TreeNode

└ **BTreeNode**

Each B-tree node has a sorted array *data* of the elements held within it, an array of child references, a parent reference. Each node also holds the parent index most recently used to access this node. We have chosen to use a sorted array for holding the elements, since it is space efficient, and it enables a binary search to be used to efficiently find an element, or the child reference to follow in a search. To avoid any costs associated with resizing an array, both the *data* and *children* arrays are allocated to accommodate a full node. The drawback of having a sorted array, is that it takes $O(t)$ time to insert an element in the middle of an array. Since a B-tree is designed to minimize space usage, it is crucial that as many elements as possible can fit on a disk page. Also, it is important that the search time for an element be minimized. Thus the alternative solution of a linked list is not good for this application, since it requires more space, and would increase the search time.

```
SortedArray<E> data;           //holds elements held in the node
BTreeNode[] children;
BTreeNode parent;
```

Several of the B-tree methods require locating the left or right sibling of a node. In order to locate these neighboring siblings, it is necessary to know the index i for which $parent.children[i] = this$. The only way to compute this value is by a linear time search within *children*. However, this value was computed during the path down to this node. To avoid the need to repeat this computation, *pIndex* holds the index of this node (within its parent's *children* array) as determined by the most recent search through this node.

```
int pIndex;                    //parent index most recently used to access this node
```

36.3.1 B-Tree Node Methods

The `BTreeNode` constructor allocates a new node with the capacity to hold $2t - 1$ elements and $2t$ children.

```
BTreeNode() {
    children = (BTreeNode[]) new BTree.BTreeNode[2*t];
    data = new SortedArray<E>(comp, 2*t-1);
}
```

The `isFrontier` method returns *true* if and only if this node is a frontier node.

```
protected boolean isFrontier(){
    return this == FRONTIER;
}
```

The `isLeaf` method returns *true* if and only if this node is a leaf node.

```
final boolean isLeaf(){
    return child(0) == FRONTIER;
}
```

Correctness Highlights: By `BALANCED` if the leftmost child is a frontier node then the node must be a leaf node. ■

The method `size` returns the number of elements held in this node.

```
protected int size() {
    return data.getSize();
}
```

Correctness Highlights: Follows from the correctness of the `SortedArray` `getSize` method. ■

The `capacity` method returns the maximum number of elements that can be held in a node.

```
protected int capacity(){
    return 2*t-1;
}
```

Correctness Highlights: By `NODEUTILIZATION` at most $2t - 1$ elements can be held in a node. ■

The `atMaxSize` method returns *true* if and only if this node is full.

```
final boolean atMaxSize(){
    return (size() == 2*t-1);
}
```

Correctness Highlights: By `NODEUTILIZATION` at most $2t - 1$ elements can be held in a node. The rest of the correctness follows from that of `size`. ■

The `atMinSize` method returns *true* if and only if this node is at the minimum size of a non-root node.

```

final boolean atMinSize(){
    return (size() == t-1);
}

```

Correctness Highlights: By NODEUTILIZATION all nodes, excluding the root, must hold at least $t - 1$ elements. The rest of the correctness follows from that of *size*. ■

The *parent* method returns the parent of this node.

```

protected BTreeNode parent() {
    return parent;
}

```

The *data* method takes i , the index of the desired element and returns the element with the given index. It throws a *PositionOutOfBoundsException* when i is not a valid index.

```

protected E data(int i) {
    if (this == AFT || this == FORE)
        throw new NoSuchElementException();
    return data.get(i);
}

```

The method *child* takes i , the index for the desired child. It returns a reference to the i^{th} child of this node. It throws an *ArrayOutOfBoundsException* when i is not between 0 and $2t - 1$.

```

protected BTreeNode child(int i) {
    children[i].pIndex = i;
    return children[i];
}

```

Correctness Highlights: This method moves to the specified child, and also updates *pIndex* to reflect the child's index within the parent. ■

The *setChild* method takes i , the index for the new child and *child*, a reference to the node to make *child[index]*. This method requires that the given child is not already a child of another node.

```

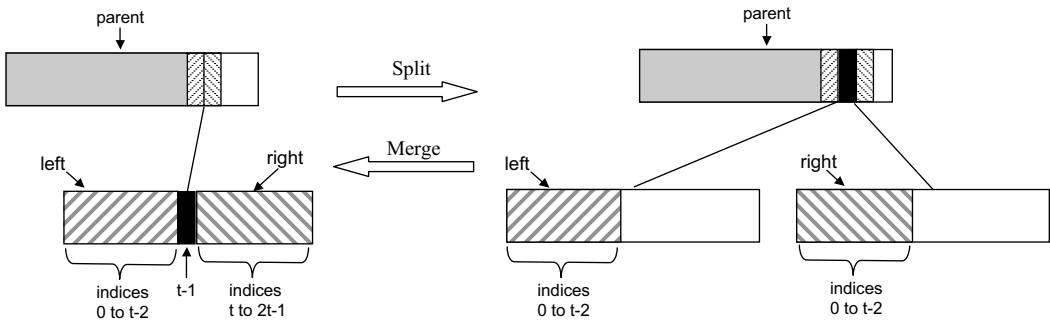
final void setChild(int i, BTreeNode child) {
    children[i] = child;
    child.parent = this;
}

```

Correctness Highlights: Whenever a child reference is set by this method, PARENT is preserved. ■

36.3.2 B-Tree Node Representation Mutators

We now present the representation mutators for the BTreeNode inner class. These methods are used to reorganize the structure of the B-tree to maintain INORDER, BALANCED, and NODEUTILIZATION. These methods serve a similar role as rotations do for a balanced binary search tree.

**Figure 36.4**

An illustration of the B-tree split and merge methods.

As illustrated in Figure 36.4 a B-tree is reorganized by splitting and merging nodes. In particular, one may take a full node (with $2t - 1$ elements) and **split** it in half with the median element being moved to the parent node, or take two minimum-sized nodes (with $t - 1$) elements and **merge** them together by pulling the element they surround from their parent. The height of a B-tree increases by one level when the root is split, and decreases by one level when the two children of a root node of size one are merged. Allowing the root to hold a single element enables these methods to be performed while preserving the B-tree properties.

To support the ability to move an element to its parent, the *addToParent* method takes *element*, the element to add to the parent of this node, and *right*, the reference to a B-tree node to add as the right child of the new element. The node on which this method is called, becomes the left child of *element*. This method creates a new root, when it is called on the root.

```
protected void addToParent(E element, BTreeNode right){
    if (this == root){                                     //splitting the root
        BTreeNode newRoot = new BTreeNode();              //create a new root
        newRoot.setChild(0, this);                          //leftmost child is this
        newRoot.addElement(0, element, right);              //add element and right child
        root = newRoot;                                     //reset root
    } else {                                                //otherwise (not splitting root)
        parent.addElement(pIndex, element, right);          //add element and right child
    }
}
```

Correctness Highlights: If this node is the root, then it has no parent. As required by the specification of this method, in this case a new root node is created with *element* as *root.data[0]*, this node as its left child, and *right* as its right child. PARENT is preserved by *setChild*.

If this node has a parent, by the correctness of the *child* method, *parent.children(pIndex) = this*. By INORDER, and the correctness of the *addElement* method, INORDER is preserved. ■

The *split* method splits this B-tree node by moving the median element to the parent. It returns a new B-tree node that is created to hold the elements right of the median. The *split* method requires that the node on which it is called is a full node.

```
BTreeNode split(){
    BTreeNode left = this;
    BTreeNode right = new BTreeNode();                      //will hold right half
```



```

    move(left, t, right, 0, t-1);           //move last t-1 elements to new node
    E moveUp = left.data.remove(t-1);      //the median element to move up to parent
    addToParent(moveUp, right);           //adjust elements and children of parent
    return right;
}

```

Correctness Highlights: By the correctness of *addToParent*, the elements in the parent are maintained in sorted order, and PARENT is preserved. Since the elements that remain in the *left* all precede *moveUp*, and the elements moved to the right all follow *moveUp* (and are kept in the same order), INORDER is maintained. Either the root splits, in which case the height grows for all paths, or the height remains unchanged. Thus BALANCED is maintained. Finally, the split node and the newly created node have $t - 1$ elements and t children, thus NODEUTILIZATION is maintained. Since all previously reachable elements remain reachable, and the children of leaf nodes are not changed, REACHABLE and FRONTIER are preserved. ■

We now describe the methods used by *remove* to restructure the B-tree. We argue that these methods preserve INORDER, BALANCED, and UTILIZATION. The *merge* method takes *parent*, the parent of the two nodes that are to be merged and *index*, the index of the child reference for this node. It merges this node with its neighboring sibling to the right. This method requires that this node, and its neighboring sibling to the right are both minimum-sized.

```

void merge(BTreeNode parent, int index) {
    BTreeNode rightSibling = parent.child(index+1);
    E moveDown = (E) parent.data.get(index);    //element from parent to move down
    parent.remove(index);                       //remove from parent
    data.add(t-1, moveDown);                    //add to this node
    move(rightSibling, 0, this, t, t-1);        //move over elements in right sibling
    if (root.size() == 0){
        root = root.child(0);                  //preserve InOrder and Reachable
        ((BTreeNode) root).parent = null;      //preserve Parent
    }
}

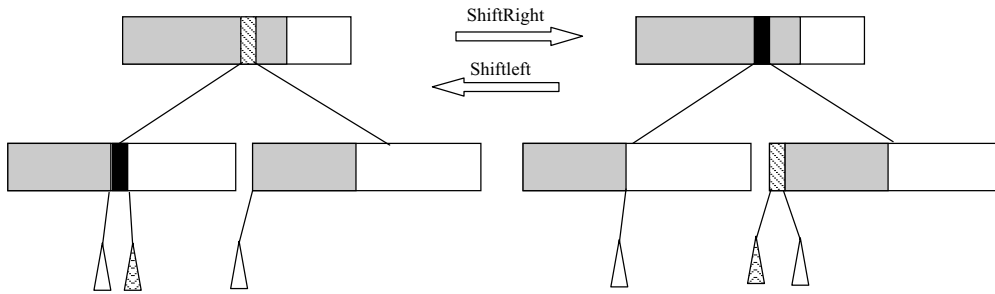
```

Correctness Highlights: By the correctness of the SortedArray *remove* methods, the elements in the parent are maintained in sorted order. Likewise, the elements in the node being merged are both in sorted order. By INORDER, $T(\text{this}) \leq \text{parent.data}(\text{index}) \leq T(\text{rightSibling})$, thus the elements are placed in this node in sorted order maintaining INORDER. If *parent* holds just one element, then the height decreases by 1 for all paths. Otherwise, the height remains unchanged. Thus BALANCED is preserved. Finally, by the requirement of the *merge* method this node and its right sibling have $t - 1$ elements. Combined with element *moveDown*, at the end of *merge*, this node holds $2(t - 1) + 1 = 2t - 1$ elements. So NODEUTILIZATION is maintained.

Finally, if the size of the root has been reduced to zero, then the only child of the current root (that at index 0) becomes the new root. ■

To insert a new element, just the *split* and *add* methods are needed. When removing elements, the *merge* method cannot be applied unless both of the left and right children surrounding a given element are at their minimum size. The other option is to either shift an element from the right child to left child (*shiftLeft*), or from the left child to the right child (*shiftRight*). Figure 36.5 illustrates these methods. Observe that these both preserve INORDER, BALANCED, and NODEUTILIZATION.

The *shiftLeft* method takes *parent*, the parent of this node, and *i*, the index of this node in its parent's *children* array. It requires that the right sibling of this node is not minimum sized, and that this node is not full.

**Figure 36.5**

This diagram illustrates the B-tree shift left and shift right methods. The unshaded portions are unused. The shaded child is the only one that is moved. All other children are unchanged. For illustrative purposes just two of the unchanged children are shown.

```

void shiftLeft(BTreeNode parent, int i) {
    BTreeNode left = this;
    BTreeNode right = parent.child(i+1);
    E moveDown = (E) parent.data.get(i);           //element to move down to left
    left.data.add(size(), moveDown);                //add moveDown to end of left
    E moveUp = (E) right.data.remove(0);            //element to move up to parent
    parent.data.set(i, moveUp);                     //move into parent
    left.setChild(size(), (BTreeNode) right.leftmostChild()); //move subtree and
    System.arraycopy(right.children, 1, right.children, 0, right.size()-1); //right's children
    right.children[right.size()-1] = null;          //reset child in right not in use to null
}

```

Correctness Highlights: By INORDER, $T(\text{child}(i)) \leq \text{data}(i) \leq T(\text{child}(i+1))$. Thus, moving the leftmost element of the right child, *moveUp* preserves that the elements in the parent are in sorted order. Similarly, inserting *moveDown* = *data*(*i*) at the end of the data in the left child keeps the left child in sorted order. Finally, moving the leftmost child from the right subtree to be the rightmost child of the left subtree, preserves INORDER. The height of all subtrees does not change so BALANCED is preserved. Finally, by the requirements of the method, NODEUTILIZATION is preserved. ■

The *shiftRight* method takes *parent*, the parent of this node, and *i*, the index of this node in its parent's *children* array. This method requires that this node is not full, and that its left sibling is not minimum sized.

```

void shiftRight(BTreeNode parent, int i) {
    BTreeNode left = parent.child(i-1);
    BTreeNode right = this;
    BTreeNode moveToRight = (BTreeNode) left.rightmostChild();
    E moveDown = (E) parent.data.get(i-1);           //element to move down to right
    right.data.add(0, moveDown);                     //add moveDown to end of left
    E moveUp = (E) left.data.remove(left.size()-1);  //element to move up to parent
    parent.data.set(i-1, moveUp);                    //move into parent
    System.arraycopy(right.children, 0, right.children, 1, right.size()-1); //right's children
    right.setChild(0, moveToRight);                  //move subtree from left to right
}

```

```

        left.children[left.size()+1] = null;           //reset child in left not in use to null
    }

```

Correctness Highlights: By INORDER, $T(\text{child}(i-1)) \leq \text{data}(i-1) \leq T(\text{child}(i))$. The rest is symmetric to that for *shiftLeft*. ■

36.3.3 B-Tree Node Content Mutators

The *addElement* method adds an element to a B-tree node. Since the number of children is always one more than the number of elements, whenever an element is added to a node, a child must also be added. We use the convention that new child is added as the right child of the given element. Specifically, the *addElement* method takes *i*, the index where the new element is to be added, *element*, the element to add, and *rightChild*, the right child for the new element.

```

BTreeNode addElement(int i, E element, BTreeNode rightChild){
    BTreeNode x = this;           //pointer to node where element will be added
    if (atMaxSize()) {             //if node is full split
        BTreeNode newNode = split();
        if (i ≥ t) {               //if it was on the right side
            i -= t;                //adjust index and
            x = newNode;           //the node to add to
        }
    }
    if (x.size()-i > 0)             //need to make room for rightChild
        System.arraycopy(x.children, i+1, x.children, i+2, x.size()-i);
    x.data.add(i, element);         //add new element
    x.setChild(i+1, rightChild);    //add new child
    return x;                      //return the node where the new element was placed
}

```

Correctness Highlights: By the correctness of *atMaxSize*, when this node is full, it is split. Thus, NODEUTILIZATION is preserved. Also, observe that when the node is split, *x* is updated to reference the new node, which is where *element* is added. Furthermore, when $i \geq t$, then *newNode* is the right child from the split. Since the left child has $t - 1$ elements, and the element at position *t* was moved into the parent, the index of each element in the right child is reduced by *t*. Thus *x* and *i* continue to correspond to the node, in which to add the child, and the index where it should be added.

Since the structure of the B-tree is modified only by *split*, BALANCED is preserved. Finally, the new element and its right child are added to the specified location. ■

The *extract* method takes *i*, the index of the element to be removed, and removes it from this node. It requires that this node is not minimum sized.

```

void extract(int i) {
    data.remove(i);
}

```

The *remove* method takes *i*, the index of the element to remove. In addition to removing *this.data[i]*, the children right of the removed element are shifted to the left to maintain INORDER.

```

void remove(int i) {
    BTreeNode x = this;
    if (x.atMinSize() && x != root) {                //can't remove prior to restructuring
        BTreeNode parent = x.parent();
        if (pIndex > 0 && !parent.child(pIndex-1).atMinSize()){           //Case 1
            shiftRight(parent, pIndex);
            i++;                                //element to be removed shifted right by one
        } else if (pIndex < parent.size() &&
            !parent.child(pIndex+1).atMinSize()) {           //Case 2
            shiftLeft(parent, pIndex);
        } else {
            if (pIndex != parent.size())                     //Case 3a
                x.merge(parent, pIndex);
            else {                                           //Case 3b
                x = parent.child(pIndex-1);                //move x to left sibling
                x.merge(parent, pIndex-1);
                int offsetFromEnd = t-1-i;
                i = x.size() - offsetFromEnd;                //update to index in joined node
            }
        }
    }
    x.extract(i);
    if (x.size() - i > 0)                                //shift children right of index back by one
        System.arraycopy(x.children, i+2, x.children, i+1, x.size()-i);
    x.children[x.size()+1] = null;                        //reset child not in use to null
}

```

Correctness Highlights: The correctness of this method depends the fact that *shiftRight*, *shiftLeft*, and *merge* preserve INORDER, BALANCED, and UTILIZATION. We first consider the cases that occur when the B-tree node from which the element to be removed is at the minimum allowed size and is not the root. (This method will allow the root to be arbitrarily small. The BTree *remove* method reduces the height of the tree if the root becomes empty.) We first discuss the reorganization required to ensure that *x* is not minimum sized, and then we discuss the removal of the desired element and the adjustment of the children of *x*.

Case 1: The left sibling of this node exists, and is not minimum sized. If $pIndex > 0$ then *x* has a left sibling (which is, by definition, child $pIndex - 1$ of the parent). By the correctness of *child* and *atMinSize*, the Case 1 code is executed exactly when there is a left sibling that is not of minimum size. Thus the requirements of *shiftRight* are met, and by its correctness the given node grows by 1 while maintaining INORDER, BALANCED, and UTILIZATION. Finally, since an element is moved into the front of the given node, the index of the element to remove increases by 1.

Case 2: The right sibling of this node exists, and is not minimum sized. If $pIndex < parent.size()$, then the given node has a right sibling (which is, by definition, child $pIndex + 1$ of the parent). The rest is symmetric to Case 1, with *shiftLeft* being called instead of *shiftRight*.

Case 3: Both the left and right sibling either do not exist, or are minimum sized. This case is further broken down into the case where the given node has a right sibling, and when it does not have a right sibling.

Case 3a: The given node has a right sibling. In this case, this node is merged with its right sibling. Since the *merge* method moves the elements and children from the right sibling to the this one, the element to remove stays in the same B-tree node with the same index. The rest of the correctness follows from that of *merge*.

Case 3b: This node has no right sibling. By definition any node (excluding the root) without a right sibling, must have a left sibling. So here, this node is merged with its left sibling by calling *merge* on the element for which it is the right child. Since the *merge* method moves the elements from this node to the left sibling, the element to be removed is copied into its left sibling. Thus both *x* and *index* must be updated. There are $t - 1$ elements in the left sibling (since it is minimum sized), and one element is moved down from the parent. Thus the first element from the right sibling (originally, index 0) is placed in position t when moved. Thus *index* must be incremented by t . The rest of the correctness follows from that of *merge*.

Once these cases are handled, node *x* is not minimum sized, so the conditions of *extract* are met. After element *i* is removed, shifting its right siblings back by one position, and setting the rightmost child reference to *null*, preserves INORDER and BALANCED. ■

36.4 B-Tree Methods

In this section we present internal and public methods for the B-tree class.

36.4.1 Constructors and Factory Methods

The constructor that takes no parameters creates an empty 2-3-4 tree (i.e., a B-tree with $t = 2$) that uses the default comparator.

```
public BTree() {
    this(Objects.DEFAULT_COMPARATOR, 2);
}
```

The constructor that takes a single parameter t , the order for the B-tree creates an empty B-tree of order t .

```
public BTree(int t) {
    this(Objects.DEFAULT_COMPARATOR, t);
}
```

The constructor that takes *comp*, the function used to compare two elements, and t , the order of the B-tree, creates an empty B-tree of order t that uses the given comparator.

```
public BTree(Comparator<? super E> comp, int t) {
    super(comp);
    this.t = t;
    root = FRONTIER;
}
```

The factory method *createRoot* creates and initializes a new empty root node.

```
protected void createRoot() {
    root = new BTreeNode();
}
```

36.4.2 Algorithmic Accessors

The *find* method takes *target*, the value to search for. It returns a reference to a B-tree node that contains an occurrence of the target when there is an equivalent element in the collection. Otherwise it returns the frontier node after setting its *parent* field set to predecessor of the target on the search path. This method sets the global variable *curIndex* to hold the index for an occurrence of the target (if it is in the collection), or otherwise the insert position for the target in the last non-frontier node on the search path.

```
protected BTreeNode find(E target) {
    BTreeNode ptr = (BTreeNode) root;                                //start at the root
    while (!ptr.isFrontier()) {                                       //until a frontier node is reached
        curIndex = ptr.data.find(target);                             //find the element in the node
        if (curIndex < ptr.size() && comp.compare(target, ptr.data.get(curIndex)) == 0)
            return ptr;                                              //return the ptr to the current node
        ((BTreeNode) FRONTIER).parent = ptr;                         //set frontiers' parent to ptr
        ptr = ptr.child(curIndex);                                    //go to the appropriate child
    }
    return ptr;                                                      //not found, return ptr to frontier node with parent set
}
```

Correctness Highlights: By REACHABLE and INORDER, this method is guaranteed to find the target if it is in the collection. By FRONTIER and the correctness of *isFrontier*, termination is guaranteed. Finally, by INUSE, no comparison involves a frontier node. ■

The internal method *getLastNodeSearchIndex* returns the index, within its tree node, of the element returned in the most recent search. This method requires that it is called only after a successful search.

```
protected int getLastNodeSearchIndex() {
    return curIndex;
}
```

The *findFirstInsertPosition* method takes *target*, the target element. It returns a reference to the B-tree node where the target would be inserted to precede any equivalent elements in the iteration order. It sets the global variable *curIndex* to the target's insert position within the returned node.

```
BTreeNode findFirstInsertPosition(E target) {
    BTreeNode ptr = (BTreeNode) root;                                //start at the root
    BTreeNode returnNode = null;
    int returnIndex = 0;
    while (!ptr.isFrontier()) {                                       //until a frontier node is reached
        curIndex = ptr.data.findFirstInsertPosition(0, ptr.size()-1, target);
        if (curIndex < ptr.size() && comp.compare(target, ptr.data.get(curIndex)) == 0) {
            returnNode = ptr;                                         //remember node and
        }
    }
```

```

        returnIndex = curIndex;                                //index where found
    }
    ((BTreeNode) FRONTIER).parent = ptr;                       //set frontier's parent to ptr
    ptr = ptr.child(curIndex);
}
if (returnNode == null)                                       //no equivalent element found
    return ptr.parent;
curIndex = returnIndex;                                       //use saved values for index and node
return returnNode;
}

```

Correctness Highlights: The correctness argument is similar to that of *find*, except that the SortedArray *findFirstInsertPosition* method is used instead of the *find* method. Also, the search does not end when a node is found holding the target. Instead, the value of *ptr* and *curIndex* for the first occurrence in the given node are stored in *returnNode* and *returnIndex*, respectively. Then the search continues at the left child of the first occurrence of the element.

When the while loop terminates, *returnNode* is *null* if and only if there is no element equivalent to the target. In this case, the search path would have continued at *children[curIndex]* of *FRONTIER.parent*. By INORDER, it follows that inserting the target at index *curIndex* of *FRONTIER.parent* would place it in the iteration order in sorted order.

When *returnNode* is not *null*, it is known that the first occurrence of the target in the iteration order is at *returnNode.data[curIndex]*. Observe that inserting the target in this position would place it in the iteration order just before any equivalent elements. ■

The *findLastInsertPosition* method takes *target*, the target element. It returns a reference to the B-tree node where the target would be inserted to follow any equivalent elements in the iteration order. It sets the global variable *curIndex* to hold the insert position for the target in the returned node.

```

BTreeNode findLastInsertPosition(E target) {
    BTreeNode ptr = (BTreeNode) root;                          //start at the root
    BTreeNode returnNode = null;
    int returnIndex = 0;
    while (!ptr.isFrontier()) {                                  //until a frontier node is reached
        curIndex = ptr.data.findLastInsertPosition(0, ptr.size()-1, target);
        ((BTreeNode) FRONTIER).parent = ptr;                  //set frontiers' parent to ptr
        if (curIndex < ptr.size() && comp.compare(target, ptr.data.get(curIndex)) == 0) {
            returnNode = ptr;
            returnIndex = curIndex+1;
            ptr = ptr.child(returnIndex);
        }
        else
            ptr = ptr.child(curIndex);
    }
    if (returnNode == null)                                     //no equivalent element found
        return ptr.parent;
    else {
        curIndex = returnIndex;                                 //use saved values for index and node
        return returnNode;
    }
}

```

Correctness Highlights: Symmetric to *findFirstInsertPosition*, the SortedArray *findLastInsertPosition* method is used. Also, if the element is found, then the right child of that element is followed. ■

The *pred* method takes *x*, a reference to a B-tree node holding the target element, and *index*, the index of the target element within the parent. It returns a reference to the B-tree node holding the predecessor of the target element, or FORE if it has no predecessor. This method sets the global variable *curIndex* to hold the index of the predecessor of the target element, if it exists.

```
TreeNode pred(BTreeNode x, int index) {
    BTreeNode ptr = x;
    if (!x.isLeaf()) {                                     //Case 1
        ptr = (BTreeNode) rightmost(x.child(index));
        curIndex = ptr.size()-1;
    } else if (index > 0) {                                //Case 2a
        curIndex = index-1;
    } else if (x == root) {                                //Case 2b
        return FORE;
    } else {                                               //Case 2c
        BTreeNode parent = (BTreeNode) x.parent();
        while (parent != root && parent.child(0) == ptr) { //move up tree as long
            ptr = parent;                                     //as ptr is a leftmost
            parent = (BTreeNode) ptr.parent();               //child of parent
        }
        curIndex = ptr.pIndex;                             //reset global var curIndex
        ptr = parent;
        if (ptr == root && curIndex == 0)
            return FORE;
        else
            curIndex--;
    }
    return ptr;
}
```

Correctness Highlights: Let $e = x.data[index]$. As in a binary search tree, the predecessor is defined structurally as the element in the inorder traversal immediately before it. We consider the following cases.

Case 1: x is not a leaf. By INORDER, the predecessor of e is the maximum child of the left subtree for e . The remainder of the correctness for this case follows from the abstract search tree *rightmost* method.

Case 2: x is a leaf. This case is further broken down into the following three subcases.

Case 2a: e is not the leftmost element in x . By INORDER, the predecessor for e is the element just before it in node x . This is reflected by leaving *ptr* unchanged and decrementing *curIndex*.

Case 2b: e is leftmost element of single node B-tree. By INORDER, e is the minimum element in the collection and thus it has no predecessor, so FORE is returned.

Case 2c: e is the leftmost element of non-singleton leaf node x . Let node y be the lowest ancestor of x , for which x is not in the leftmost subtree. Observe that after executing $ptr = parent$ (just after the *while* loop terminates), $ptr = y$. Let C_i be the child of y that was taken on the path to x . By definition, $i > 0$ since it is not the leftmost child. There are two cases that can occur. Either y does not exist since the leftmost child is always taken. In this case, by INORDER, e is the minimum element and has no predecessor, so FORE is the correct return value. We now consider when y exists. By INORDER, the predecessor of e is the element in y whose right child is C_i . That is, the predecessor of e is $y.data[i-1]$. By decrementing $curIndex$, element $i - 1$ is properly returned. ■

The *predecessor* method takes *target*, the element for which to find the predecessor. It returns the largest element in the ordered collection that is less than *target*. This method does not require that *target* be in the collection. It throws a *NoSuchElementException* when no element in the collection is less than *target*.

```
public E predecessor(E target) {
    BTreeNode x = findFirstInsertPosition(target);           //curIndex is set
    if (x.isLeaf() && curIndex > 0)                          //if insert position not left of first element
        return x.data[curIndex-1];
    return pred(x, curIndex).data[curIndex];                //curIndex is updated by pred
}
```

Correctness Highlights: Observe that the predecessor of any element would precede its first insert position. For efficiency, if the first insert position is an element of a leaf other than the leftmost element, then the predecessor directly precedes it in that leaf. The rest of the correctness follows from the correctness of *findFirstInsertPosition* and *pred*. ■

The *succ* method takes x , a reference to a B-tree node holding the target element, and *index*, the index of the target element. It returns a reference to the B-tree node holding the successor of the target element, or AFT if it has no successor. This method sets the global variable *curIndex* to hold the index of the predecessor of the target element, if it exists.

```
TreeNode succ(BTreeNode x, int index){
    BTreeNode ptr = x;
    if (!x.isLeaf()){                                       //Case 1
        ptr = (BTreeNode) leftmost(x.child(index+1));
        curIndex = 0;
    } else if (index < x.size() - 1){                      //Case 2a
        curIndex = index+1;
    } else if (x == root){                                 //Case 2b
        return AFT;
    } else {                                               //Case 2c
        BTreeNode parent = ptr.parent();
        while (parent != root && parent.child(parent.size()) == ptr) { //move up tree as long
            ptr = ptr.parent();                                //as ptr is a rightmost
            parent = ptr.parent();                             //child of parent
        }
        curIndex = ptr.pIndex;                             //reset global variable curIndex
        ptr = parent;
        if (ptr == root && curIndex == ptr.size())
            return AFT;
    }
}
```

```

        return AFT;
    }
    return ptr;
}

```

Correctness Highlights: The correctness mirrors that of the *pred* method. We briefly summarize the cases. Let $e = x.data[index]$.

Case 1: x is not a leaf. Then the successor is the minimum child of the right subtree of e . The remainder of the correctness for this case follows from the abstract search tree *leftmost* method.

Case 2: x is a leaf. This case is further broken down into the following three subcases.

Case 2a: e is not the rightmost element in x . By IN ORDER, the successor for e is the element just after it in node x .

Case 2b: e is rightmost element of single node BTree. By INORDER, e is the maximum element in the collection, so AFT is correctly returned.

Case 2c: e is the rightmost element of leaf node x . Symmetric to Case 2c for *pred*. ■

The *successor* method takes *target*, the element for which to find the successor. It returns the least element in the ordered collection that is greater than *target*. This method does not require that *target* be in the collection. It throws a *NoSuchElementException* when no element in the collection is greater than *target*.

```

public E successor(E target) {
    BTreeNode x = findLastInsertPosition(target);           //curIndex is set
    if (curIndex < x.size())                                //if insert position not after last element
        return x.data(curIndex);
    return succ(x, curIndex).data(curIndex);                //curIndex is updated by succ
}

```

Correctness Highlights: As long as the index for the last insert position is not after the last element in the node x returned from *findLastInsertPosition*, then $x.data(curIndex)$ is the successor. Otherwise, the successor is the first element in the next node in the inorder traversal of the tree. By the correctness of *succ*, this will be the value returned. ■

36.4.3 Representation Mutators

The *move* method takes *from*, a reference to the B-tree node from which elements are to be moved, *fromIndex*, the index of the first element to be moved, *to*, a reference to the B-tree node where the elements are to be moved, *toIndex*, the destination index for the first element, and *num*, the number of elements (and their surrounding children) to be moved. This method assumes that the calling method maintains NODEUTILIZATION.

```

void move(BTreeNode from, int fromIndex, BTreeNode to, int toIndex, int num){
    for (int i = 0; i < num; i++){

```

```

        to.data.add(toIndex+i, from.data.remove(fromIndex));
        to.setChild(toIndex+i, from.child(fromIndex+i));
    }
    to.setChild(toIndex+num, from.child(fromIndex+num));           //move last child
    Arrays.fill(from.children, fromIndex, fromIndex+num+1, null);
}

```

36.4.4 Content Mutators

Methods to Perform Insertion

There are two possible strategies for inserting a new element into a B-tree. The first approach is **top-down insertion**, in which any full node encountered during the original search is split (even though it might not be necessary). This approach guarantees that when a leaf node is reached, it cannot be full. Thus the new element can just be added in the leaf node where the search for it ends. Top-down insertion is designed for the situation in which each node is stored on a disk page. It avoids the need for a possible pass back up the tree, which guarantees that no disk page must be fetched more than once during the insertion process.

In **bottom-up insertion**, the new element is inserted at the leaf node where the search for it ends. If that leaf node is not full then nothing more must be done. Otherwise, the node is split. However, when a node is split, an additional element must be added to its parent. If the parent is not full then a single split is sufficient. However, in general, the process of splitting can propagate up the tree to the root. The advantage of this approach is it minimizes the restructuring done. So for a 2-3-4 tree (i.e., when $t = 2$), a bottom-up approach is preferred. Even when secondary storage is used, in general the cache size is sufficient that all pages on the search path can fit in the cache, so no page faults are incurred during the pass back up the tree.

The public *add* method is inherited from *AbstractSearchTree*. It uses the internal *insert* method that takes *element*, the new element, and inserts it into the tree using the bottom-up insertion approach. The *insert* method returns a reference to the newly added B-tree node.

```

protected TreeNode insert(E element){
    if (isEmpty()) {
        createRoot();
        ((BTreeNode) root).data.add(0, element);
        ((BTreeNode) root).children[0] = (BTreeNode) FRONTIER;
        ((BTreeNode) root).children[1] = (BTreeNode) FRONTIER;
        return root;
    } else {
        BTreeNode ptr = (BTreeNode) root;           //start search at the root
        while (!ptr.isLeaf()) {                       //until a frontier node is reached
            curIndex = ptr.data.findLastInsertPosition(0, ptr.size()-1, element);
            ptr = ptr.child(curIndex);
        }
        curIndex = ptr.data.findLastInsertPosition(0, ptr.size()-1, element);
        version.increment();                          //invalidate all markers for iteration
        return ptr.addElement(curIndex, element, (BTreeNode) FRONTIER);
    }
}

```

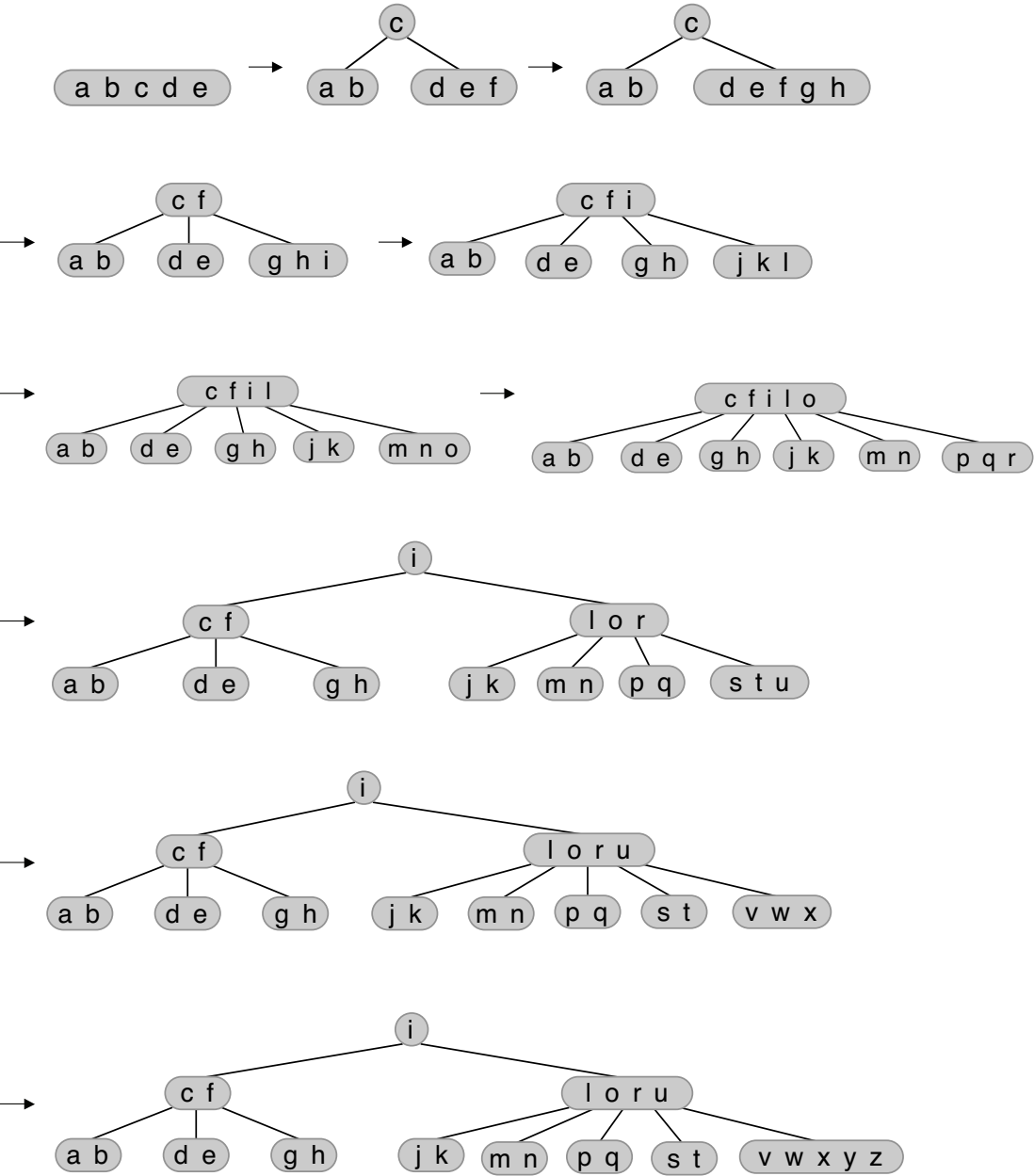


Figure 36.6
A B-tree insertion example with $t = 5$ obtained by inserting the letters of the alphabet, in alphabetical order.

Correctness Highlights: The inherited public *insert* method preserves SIZE. It is easily verified that when the collection is empty, the method creates a B-tree that adheres to all properties. By INORDER the leaf node where *element* should be inserted is obtained. (When there are duplicates, we have chosen to insert the new element after all equivalent elements in the iteration order.) The rest of the correctness follows from the BTreeNode *add* method. ■

If a top-down insertion is preferred, then the while loop in the *insert* method would be replaced by:

```

while (ptr.atMaxSize() || !ptr.isLeaf() ) {
    curIndex = ptr.data.findLastInsertPosition(0, ptr.size()-1, element);
    if (ptr.atMaxSize()){
        BTreeNode newNode = ptr.split();
        if (curIndex ≥ t) {
            curIndex -= t;
            ptr = newNode;
        }
    }
    else
        ptr = ptr.child(curIndex);
}

```

Observe that in a top-down insertion, whenever an element is added to a node, that node is guaranteed to not be full. So when using top-down insertion, the BTreeNode *add* can be modified to require that the node on which it is called is not full.

Methods to Perform Deletion

We now describe the *remove* method that takes *x*, a reference to the B-tree node holding the element to remove, and *index*, the index in *x* of the element to remove. This method removes $e = x.data[index]$ using an approach like that for a binary search tree. As with insertion, we use a bottom-up approach. If *x* is a leaf node then *e* is removed from *x*. (Recall that the BTreeNode *remove* method restructures the tree if the node from which the element to be removed is minimum-sized.) If *x* is an internal node, then the predecessor e_p of *e* replaces *e*, and then e_p , which is guaranteed to be in a leaf, is removed from that leaf. Figure 36.7 illustrates an execution of *remove*. Observe, that during the execution of *remove*, once e_p replaces *e*, it occurs in the B-tree twice until the removal of e_p from its leaf position is completed.

```

void remove(BTreeNode x, int index) {
    if (x.isLeaf()) {
        x.remove(index);
        size--;
    } else {
        BTreeNode pred = (BTreeNode) rightmost(x.child(index));
        x.data.set(index, pred.data(pred.size()-1));
        remove(pred, pred.size()-1);
    }
}

```

//Case 1
//Case 2

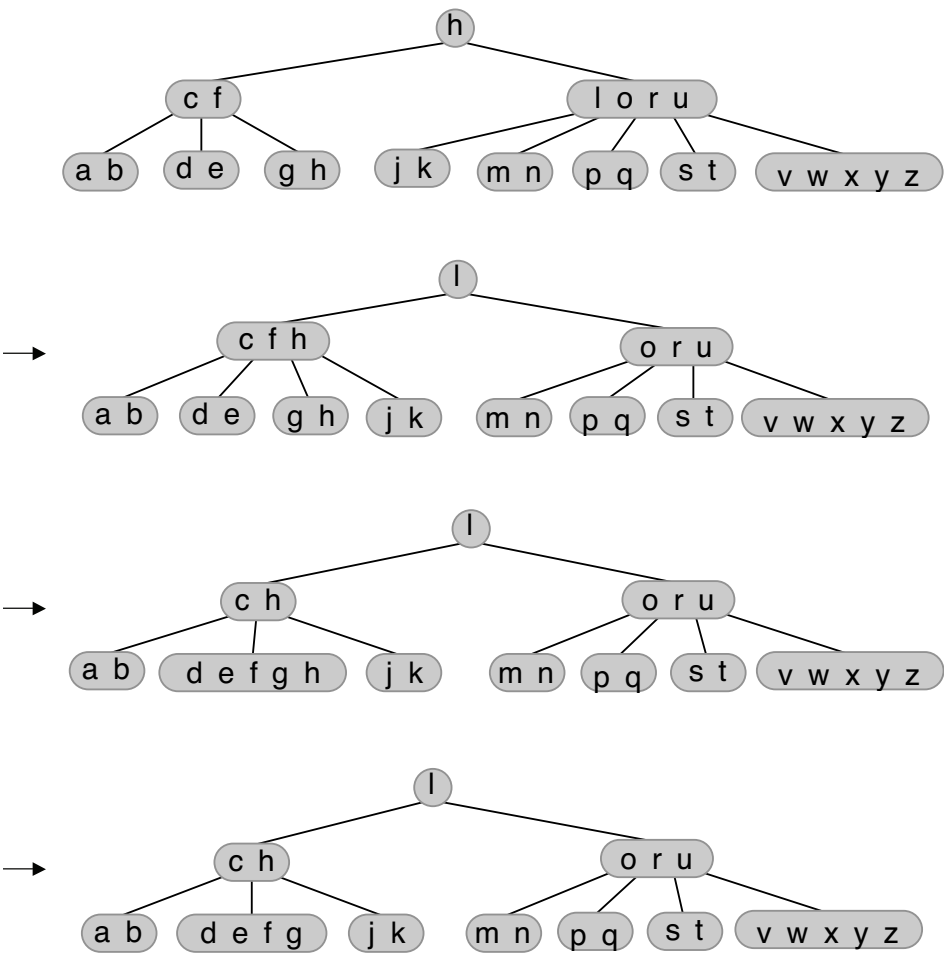


Figure 36.7
An illustration of the steps to remove “i” from the B-tree of Figure 36.6. First “i” is replaced by its predecessor “h” (which occurs twice until the *remove* method is completed). Then a recursive call is made to remove the “h” in the leaf. First *shiftLeft* is performed at the root level. Next *merge* is performed at the middle level. Finally, “h” is removed from the leaf, which is no longer at its minimum size.

Correctness Highlights: Let e be the element to be removed. There are two cases that can occur. Recall that the `BTreeNode remove` method preserves *InOrder*, *Balanced*, and *NodeUtilization*.

Case 1: x is a leaf. In this case, the `BTreeNode remove` method can be called.

Case 2: x is a non-leaf. In this case, the element that precedes e in the iteration order is located. By *INORDER*, this element is guaranteed to be the rightmost node in the left subtree. Replacing the element to remove by its predecessor preserves *INORDER*. Then `remove` is recursively called to remove this element, which by definition is the rightmost element in *pred*. ■

If a top-down remove method is preferred, then on the search down to the element to remove, whenever a non-root node at its minimum size is encountered, reorganization like that in the `BTreeNode remove` method must be executed. Similarly, if the element e to remove is in an internal node, when a minimum-sized node is encountered on the path to e 's predecessor, the tree must be reorganized. Care must be taken to ensure that the location of e is maintained during any reorganization so that it can be replaced by its predecessor. Since a top-down remove always ensures the next node in the search path is not minimum sized before moving to it, the `BTreeNode remove` method can assume that the node on which it is called is not minimum sized. The `TopDownBTree` class on the CD is an extension of the `BTree` class that uses top-down insertion and deletion.

The public `remove` method is inherited from `AbstractSearchTree`. It uses the following internal `remove` method that takes x , a reference to an existing tree node, and removes the element at *curIndex* of x .

```
protected void remove(TreeNode x) {
    remove((BTreeNode) x, curIndex);           //curIndex was set by find in public remove
    version.increment();                         //invalidate all markers for iteration
}
```

The `clear` method removes all elements from the collection. Since it is an untracked implementation, there is no need to iterate through the elements.

```
public void clear(){
    root = FRONTIER;
    size = 0;
    version.increment();                         //invalidate all markers for iteration
}
```

36.4.5 Locator Initializers

The `iterator` method creates a new tracker that is at *FORE*.

```
public Locator<E> iterator() {
    return new Marker((BTreeNode) FORE, -1);
}
```

The `iteratorAtEnd` method creates a new tracker that is at *AFT*.

```

public Locator<E> iteratorAtEnd() {
    return new Marker((BTreeNode) AFT, -1);
}

```

The method *getLocator* takes *x*, the element to track. It returns a locator initialized at *x*. It throws a *NoSuchElementException* when there is no element equivalent to *x* in the collection.

```

public Locator<E> getLocator(E x) {
    BTreeNode t = find(x);
    if (t == FRONTIER)
        throw new NoSuchElementException();
    return new Marker(t, curIndex);
}

```

36.5 Marker Inner Class

`AbstractCollection<E>.AbstractLocator<E>` implements *Locator<E>*

↳ **Marker** implements *Locator*

The B-tree marker throws a concurrent modification exception when any mutation is made. Thus, it is an untracked implementation with *remove* not supported. Each Marker element has two instance variables, *node* a reference to the node holding the tracked element, and *index*, the index of the tracked element within *node*. The *index* is -1 for FORE and AFT.

```

BTreeNode node;
int index;

```

The constructor takes two arguments, *x*, a reference the node to track, and *index*, the index of the element to track in *x*.

```

Marker(BTreeNode x, int index) {
    node = x;
    this.index = index;
}

```

Recall that *inCollection* returns *true* if and only if the tracked element is currently in the collection.

```

public boolean inCollection() {
    checkValidity();
    return (node != FORE && node != AFT);
}

```

Correctness Highlights: The only location in which the marker is not at an element in the collection is at FORE and AFT. (An equivalent condition would be to return *index != -1* since the index is only -1 for FORE and AFT.) ■

The *get* method returns the marked element. It throws a *NoSuchElementException* when marker is not at an element in the collection.


```

public E get() {
    checkValidity();
    return node.data(index);
}

```

The *advance* method moves the tracker to the next element in the iteration order, or AFT if the last element is currently tracked. It returns *true* if and only if after the update, the tracker is at an element of the collection. It throws an *AtBoundaryException* when the tracker is at AFT since there is no place to advance.

```

public boolean advance() {
    checkValidity();
    if (node == AFT)
        throw new AtBoundaryException();
    if (node == FORE && isEmpty())
        node = (BTreeNode) AFT;
    else if (node == FORE) {
        node = (BTreeNode) leftmost();
        index = 0;
    } else {
        node = (BTreeNode) succ(node, index);
        index = curIndex;
    }
    return node != AFT;
}

```

*//when the marker is at FORE
//first element in iteration order
//is element 0 of the leftmost node

//move to successor

//still within collection unless AFT reached*

Correctness Highlights: If the marker is at AFT, the exception is properly thrown. If the marker is at FORE, then by INORDER the next element in the iteration order is the first element in the leftmost node of the B-tree (unless the collection is empty), and the rest of the case follows from the correctness of *leftmost*. We now consider when the marker is at neither FORE nor AFT. By the correctness of *succ*, the marker is moved to the next element in the iteration order. Finally, after updating the marker location, if the marker is not at AFT, then it is at an element in the collection, so the correct value is returned. ■

The *retreat* method moves the tracker to the previous element in the iteration order, or FORE if the first element is currently tracked. It returns *true* if and only if after the update, the tracker is at an element of the collection. It throws an *AtBoundaryException* when the tracker is at FORE since then there is no place to retreat.

```

public boolean retreat() {
    checkValidity();
    if (node == FORE)
        throw new AtBoundaryException();
    if (node == AFT && isEmpty())
        node = (BTreeNode) FORE;
    else if (node == AFT) {
        node = (BTreeNode) rightmost();
        index = node.size()-1;
    } else {
        node = (BTreeNode) pred(node, index);
    }
}

```

*//otherwise, if tracker is at AFT
//go to last element

//otherwise, move to the predecessor*

```

        index = curIndex;
    }
    return node != FORE;           //still within collection unless FORE reached
}

```

Correctness Highlights: If the marker is at FORE, the exception is properly thrown. If the marker is at AFT, then by INORDER the next element in the iteration order is the last element in the rightmost node of the B-tree (unless the collection is empty), and the rest follows from the correctness of *rightmost*. We now consider when the marker is neither at FORE nor AFT. By the correctness of *pred*, the marker is moved to the next element in the iteration order. Finally, after updating the marker location, if the marker is not at FORE, then it is at an element in the collection, so the correct value is returned. ■

The *hasNext* method returns *true* if there is some element after the current locator position.

```

public boolean hasNext() {
    checkValidity();
    if (node == FORE)
        return !isEmpty();
    if (node == AFT)
        return false;
    return succ(node, index) != AFT;
}

```

Correctness Highlights: If the marker is at FORE, then there is another element in the collection as long as the collection is not empty. If the marker is at AFT, then there is not an element after it in the iteration order. Otherwise, by the correctness of *succ*, there is another element in the iteration order as long as the successor of the current marker location is not AFT. ■

Since it would be expensive to update the marker when an element is removed, the *remove* method is not supported. If desired, the *getLocator* method could be used to reset the marker.

```

public void remove() {
    throw new UnsupportedOperationException();
}

```

36.6 Performance Analysis

The asymptotic time complexity of each public method of the BTree class is given in Table 36.8, and the asymptotic time complexity for each public method of the BTree.Marker class is given in Table 36.9. For a B-tree node x , we let h_x be defined as the length of the search path from the root to x . The **height** of a B-tree is defined as height of the leaf nodes, which, by BALANCED, is the same for all leaves. For example, in the B-tree of Figure 36.1 the height is 2. We also compute an upper bound on the number of disk pages read to locate the target element. We assume that the root is kept in memory.

We start by computing the minimum number of elements that can be in a B-tree of height h . By NODEUTILIZATION, the minimum number of elements occurs when the root holds one element (so

method	worst-case time complexity	worst-case disk pages read
constructors	$O(1)$	0
<i>iterator()</i>	$O(1)$	0
<i>iteratorAtEnd()</i>	$O(1)$	0
<i>ensureCapacity(x)</i>	$O(1)$	0
<i>trimToSize()</i>	$O(1)$	0
<i>min()</i>	$O(\log_t n)$	$\log_t n$
<i>max()</i>	$O(\log_t n)$	$\log_t n$
<i>contains(o)</i>	$O(\log_2 n)$	$\log_t n$
<i>getEquivalentElement(o)</i>	$O(\log_2 n)$	$\log_t n$
<i>getLocator(o)</i>	$O(\log_2 n)$	$\log_t n$
<i>predecessor(o)</i>	$O(\log_2 n)$	$\log_t n$
<i>successor(o)</i>	$O(\log_2 n)$	$\log_t n$
<i>remove(o)</i>	$O(t \log_t n)$	$2 \log_t n$
<i>add(o)</i>	$O(t \log_t n)$	$2 \log_t n$
<i>accept(v)</i>	$O(n)$	$n/(t-1)$
<i>toArray()</i>	$O(n)$	$n/(t-1)$
<i>toString()</i>	$O(n)$	$n/(t-1)$
<i>retainAll(c)</i>	$O(n c + n \log_t n)$	$2 c \log_t n$
<i>addAll(c)</i>	$O(c \log_t(n + c))$	$2 c \log_t(n + c)$

Table 36.8 Summary of the asymptotic time complexities for the OrderedCollection public methods when using the B-tree data structure. We assume that no page faults occur in accessing collection c for *addAll* and *retainAll*.

locator method	worst-case time complexity	worst-case disk pages read
constructor	$O(1)$	0
<i>get()</i>	$O(1)$	1
<i>advance()</i>	$O(\log_2 n)$	$\log_t n$
<i>retreat()</i>	$O(\log_2 n)$	$\log_t n$
<i>next()</i>	$O(\log_2 n)$	$\log_t n$
<i>hasNext()</i>	$O(\log_2 n)$	$\log_t n$

Table 36.9 Summary of the asymptotic time complexities for the public methods of the BTree Marker class.

2 children) and all other nodes hold $t - 1$ elements (t children). Thus

$$\begin{aligned}
 n &\geq 1 + 2(t - 1) + 2t(t - 1) + 2t^2(t - 1) + \dots + 2t^{h-1}(t - 1) \\
 &= 1 + 2(t - 1) \sum_{i=0}^{h-1} t^i \\
 &= 1 + (2(t - 1)(t^h - 1))/(t - 1) \\
 &= 1 + 2(t^h - 1) \\
 &= 2t^h - 1
 \end{aligned}$$

Solving for h yields that $t^h \leq (n + 1)/2$, so

$$h \leq \log_t((n + 1)/2) \approx \log_t n.$$

Clearly the constructors take constant time. Likewise, the locator constructor runs in constant time, so *iterator* and *iteratorAtEnd* take constant time. Also, since the B-tree is an elastic implementation, *ensureCapacity* and *trimToSize* need not perform any computation, so they take constant time. Since the B-tree is not tracked, *clear* can be performed in constant time.

The time to locate the target element, or its insert position, involves a single pass down the tree, where $O(\log_2 t)$ time is spent at each level for the binary search. Observe that

$$\log_2 t \cdot \log_t n = \log_2 t \cdot \frac{\log_2 n}{\log_2 t} = \log_2 n.$$

Thus the time to locate the target, or its insertion position, is $O(\log_2 n)$. The number of disk pages read is at most $h = O(\log_t n)$. Thus *contains*, *getEquivalentElement*, and *getLocator* take $O(\log_2 n)$ time.

The *min* method takes a single pass down the tree, always going to the left child, and the *max* method takes a single pass down the tree, always going to the right child. Thus both take $O(h) = O(\log_t n)$ time.

The *BTreeNode.split* method takes $O(t)$ time to move the last $t - 1$ elements to the new node. As discussed above, the search performed within the *add* method takes $O(\log_2 t)$ time. In the worst case, when all nodes on the search path are full, the *add* method must split $O(h)$ nodes leading to a cost of

$$O(\log_2 n + t \log_t n) = O(\log_2 t \cdot \log_t n + t \log_t n) = O(t \log_t n).$$

Likewise, the *merge* method used in the B-tree *remove* takes $O(t)$ time. The search performed within *remove* takes $O(\log_2 t)$ time, and in the worst-case all nodes on the search path are at their minimum size and need to be split. So *remove* has worst-case cost $O(t \log_t n)$. Since both *add* and *remove* may need to examine an extra B-tree node at each level, up to $2 \log_t n$ disk pages are read.

The *predecessor* and *successor* methods either take a pass up or down the tree. Thus, the worst-case time complexity for both of these methods is $O(\log_2 n)$. The number of pages read is at most the height of the tree.

The *toString* method performs an inorder traversal. Observe that during the traversal each node is visited exactly once and constant time is spent at each node. Thus, the overall cost is $O(n)$.

We conclude with time complexities of the locator methods. The methods that just access an element take $O(1)$ time. The others require the execution of either *succ* or *pred*, so each takes $O(\log_2 n)$ time.

36.7 Quick Method Reference

BTree Public Methods

- p. 557 *BTree*()
- p. 557 *BTree*(Comparator<? super E> comp, int t)
- p. 557 *BTree*(int t)
- p. 98 void **accept**(Visitor<? super E> v)
- p. 479 void **add**(E element)
- p. 100 void **addAll**(Collection<? extends E> c)
- p. 100 void **clear**()
- p. 97 boolean **contains**(E value)
- p. 99 void **ensureCapacity**(int capacity)
- p. 476 E **get**(int r)
- p. 96 int **getCapacity**()
- p. 96 Comparator<? super E> **getComparator**()
- p. 98 E **getEquivalentElement** (E target)
- p. 568 Locator<E> **getLocator**(E x)
- p. 96 int **getSize**()
- p. 96 boolean **isEmpty**()
- p. 567 Locator<E> **iterator**()
- p. 567 Locator<E> **iteratorAtEnd**()
- p. 478 E **max**()
- p. 477 E **min**()
- p. 561 E **predecessor**(E target)
- p. 479 boolean **remove**(E element)
- p. 100 void **retainAll**(Collection<E> c)
- p. 562 E **successor**(E target)
- p. 97 Object[] **toArray**()
- p. 97 E[] **toArray**(E[] array)
- p. 98 String **toString**()
- p. 99 void **trimToSize**()

BTree Internal Methods

- p. 97 int **compare**(E e1, E e2)
- p. 558 void **createRoot**()
- p. 97 boolean **equivalent**(E e1, E e2)
- p. 558 BTreeNode **find**(E target)
- p. 558 BTreeNode **findFirstInsertPosition**(E target)
- p. 559 BTreeNode **findLastInsertPosition**(E target)
- p. 476 int **getLastNodeSearchIndex**()
- p. 563 BTreeNode **insert**(E element)
- p. 477 BTreeNode **leftmost**()
- p. 476 BTreeNode **leftmost**(BTreeNode x)
- p. 562 void **move**(BTreeNode from, int fromIndex, BTreeNode to, int toIndex, int num)
- p. 560 BTreeNode **pred**(BTreeNode x, int index)
- p. 565 void **remove**(BTreeNode x, int index)
- p. 567 void **remove**(BTreeNode x)
- p. 477 BTreeNode **rightmost**()
- p. 477 BTreeNode **rightmost**(BTreeNode x)
- p. 561 BTreeNode **succ**(BTreeNode x, int index)
- p. 99 void **traverseForVisitor**(Visitor<? super E> v)
- p. 478 void **traverseForVisitor**(Visitor<? super E> v, BTreeNode x)
- p. 98 void **writeElements**(StringBuilder s)
- p. 479 void **writeElements**(StringBuilder sb)

BTree.BTreeNode Internal Methods

- p. 550** *BTreeNode*()
- p. 555** *BTreeNode* **addElement**(int i, E element, *BTreeNode* rightChild)
- p. 552** void **addToParent**(E element, *BTreeNode* right)
- p. 550** boolean **atMaxSize**()
- p. 550** boolean **atMinSize**()
- p. 550** int **capacity**()
- p. 551** *BTreeNode* **child**(int i)
- p. 551** E **data**(int i)
- p. 555** void **extract**(int i)
- p. 550** boolean **isFrontier**()
- p. 550** boolean **isLeaf**()
- p. 553** void **merge**(*BTreeNode* parent, int index)
- p. 551** *BTreeNode* **parent**()
- p. 555** void **remove**(int i)
- p. 551** void **setChild**(int i, *BTreeNode* child)
- p. 553** void **shiftLeft**(*BTreeNode* parent, int i)
- p. 554** void **shiftRight**(*BTreeNode* parent, int i)
- p. 550** int **size**()
- p. 552** *BTreeNode* **split**()

BTree.Marker Public Methods

- p. 569** boolean **advance**()
- p. 568** E **get**()
- p. 570** boolean **hasNext**()
- p. 101 void **ignoreConcurrentModifications**(boolean ignore)
- p. 101 void **ignorePriorConcurrentModifications**()
- p. 568** boolean **inCollection**()
- p. 101 E **next**()
- p. 570** void **remove**()
- p. 569** boolean **retreat**()

BTree.Marker Internal Methods

- p. 568** *Marker*(*BTreeNode* x, int index)
- p. 101 void **checkValidity**()
- p. 101 void **updateVersion**()

Chapter 37

B+-Tree Data Structure

AbstractCollection<E> implements *Collection*<E>
└─ AbstractSearchTree<E> implements *OrderedCollection*<E>
 └─ BTree<E> implements *OrderedCollection*<E>
 └─ **BPlusTree**<E> implements *OrderedCollection*<E>

Uses: Java references and sorted array (Chapter 30)

Used By: TaggedBPlusTree (Section 49.9.7)

Strengths: Like the B-tree data structure, a B+-tree is designed for ordered collections that must be held in secondary storage. As with a B-tree, a B+-tree of order t can hold up to $2t - 1$ elements in each node. By adjusting t , the B+-tree can be adapted so that a full node fills a disk page. The advantage of a B+-tree over a B-tree is that all elements in the collection are held in the leaf nodes, which are linked together to form a sorted list. Thus, a B+-tree can support constant time methods to move forward or backward in the iteration order. Having all of the elements occur in sorted order at the leaf level supports very fast **streaming access** to the data, and real-time updates of the data structure through a process known as **bulk loading**.

Weaknesses: The B+-tree requires more storage than a B-tree since the internal nodes contain duplicate elements held in the collection to use for navigation. Also, both a B-tree and B+-tree are untracked data structures that do not support removal of an element through a locator.

Critical Mutators: *add, clear, remove*

Competing Data Structures: If the ordered collection is held in main memory, consider a red-black tree (Chapter 34) or a skip list (Chapter 38). The skip list also supports a constant time method to advance or retreat in the iteration order. So if the data structure can fit in main memory, then a skip list is usually a better choice than a B+-tree. Also, it is possible, with some overhead in both space and time, to modify the red-black tree to thread the nodes into an iteration list. More specifically, two references can be added to each node, one that references the successor and one that references the predecessor. These can be updated in $O(\log n)$ time for all mutations using the logarithmic time methods to find the predecessor or successor for a node.

If the collection must be held in secondary storage but it is not important to reach the successor or predecessor in constant time, a B-tree (Chapter 36) should be considered.

When secondary storage must be used, and the size of each element is substantially larger than the portion used by the comparator, then a tagged ordered collection (Section 49.9) that wraps a B+-tree is the best option. Furthermore, if the application tends to insert many elements with the same tag, then a tagged bucket ordered collection (Chapter 50) that wraps a B+-tree should be considered.

37.1 Case Study: A Web Search Engine

In this case study, we overview the data structure support needed in a Web search engine design. Much of the material in this chapter is based upon the seminal paper of Brin and Page [31], a survey paper by Arasu et al. [12], and a book that focuses on the page rank algorithm by Langville and Meyer [100]. Before discussing the data structure design, it is important to understand some important components of the web search process.

Crawler Module: The **crawler module** creates **spiders**, small programs that browse the web to both extract URLs appearing in retrieved pages and also to retrieve pages to be stored in the page repository.

Page Repository: The **page repository** must maintain the web pages to enable fast access. Some additional requirements that affect the design of the page repository are that it must be scalable across a large cluster of computers, it must support fast access to any desired page based on its URL, and it must also support streaming insertion for new pages retrieved by the crawler module.

Indexing Module: The **indexing module** compresses each page by storing the information that is relevant for the retrieval process.

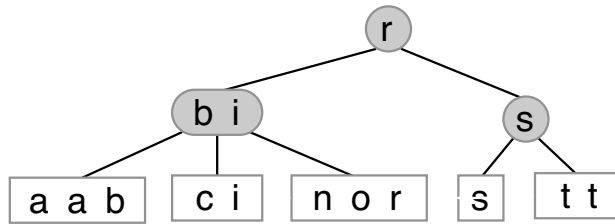
Indexes: A variety of indexes are created to support fast retrieval of the relevant pages in response to queries that are based upon the content of the page, as opposed to the page repository that organizes pages based on the URL of the page. The **content index** provides access to pages based on words that are important to the document including keywords, the title, and anchor text. The content index is stored as an **inverted file** data structure. The case study in Section 39.2 discusses support for an inverted file using a digitized ordered collection. The web can be viewed as a graph in which each page is a vertex, and a hyperlink from page *A* to page *B* is a directed edge from vertex *A* to vertex *B*. The **structure index** maintains information regarding the hyperlink structure of the pages. There are also some other special purpose indexes that are maintained (e.g., an index for images).

Query Module: The **query module** takes the user's query expressed in natural language, and uses it to access the inverted index to find pages that appear to be relevant. These pages are then passed to the ranking module.

Ranking Module: The **ranking module** uses the structure of the web to create an importance measure for each web page, and uses them to take the pages from the query module and choose a presentation order for the relevant pages based on the overall quality of these pages according to the **page rank algorithm**. To learn more about the ranking module, see the paper by Jon Kleinberg [95] and the book of Langville and Meyer [100].

For the remainder of this section, we focus on the data structure design for the page repository. Each new page retrieved by the crawler module is given a page id, in sequential order of when the page was added to the page repository. Each disk will hold a range of contiguous page ids. A very important requirement of the page repository is the ability to add new pages in real-time (**streaming access**), and also support fast access of a page by the page id (**random access**). The ability to support fast access by page id seems to indicate the use of the Mapping ADT (Section 49.7). However, since all appropriate mapping data structures are based on hashing, it cannot provide the needed real-time support for real-time insertion of new pages.

Since a B+-tree includes all elements in the leaf nodes, and the leaf nodes are linked in a sorted list, as long as the B+-tree is organized by the page id, and there is a guarantee that the new pages

**Figure 37.1**

A populated example for a B+-tree instance containing the ordered collection $\langle a, a, b, c, i, n, o, r, s, s, t, t \rangle$. The *next* and *prev* references for the leaf nodes are not shown.

are added in sorted order by page id, it provides a solution to the problem of real-time insertion of new pages. A simple variation of a B+-tree can insert each new element (which is guaranteed to be larger than the existing elements) at the end of the ordered leaf chain either by inserting it at the end of the current rightmost leaf, or creating a new rightmost leaf if the existing one is full. Then the internal nodes of the B+-tree can be added later as processing time permits. This leads to the following data structure choices for the page repository.

- A Mapping ADT held in memory is used to map from a normalized version of the URL to the corresponding web page id.
- A global tagged ordered collection (Section 49.9) is stored in memory to support finding the disk on which a given web page is stored. Each tag is the web page id of the first web page on a disk, and the associated data is the id of the disk containing that page. Each disk contains web pages for a consecutive sequence of web page ids, and these sequences do not overlap. Therefore, given a target web page id, one can search in the tagged ordered collection for a match or for the target's predecessor to find the appropriate disk id. Since this structure has only one node per disk, it is fairly small and can be held in memory, so a tagged red-black tree could be used.
- A local B+-tree holding tagged elements, where the tag is the web page id and the associated data is the information associated with the page, is stored on each disk.

37.2 Internal Representation

A B+-tree is an extension of a B-tree in which all of the elements are stored in the leaves, and the leaves are linked together in a doubly linked list. All properties of the B-tree are maintained.

Instance Variables and Constants: No additional instance variables are added. However, the constants *FORE* and *AFT* are modified to serve as the head and tail sentinels for the doubly linked list of leaf nodes.

Populated Example: Figure 37.1 shows a populated example of a B+-tree holding the letters of “abstraction,” inserted in the order they appear in the word. The unfilled rectangles are the leaf nodes, which are connected in a doubly linked list with head sentinel of *FORE*, and tail sentinel of *AFT* (not shown in the figure).

Terminology: We use the following definitions throughout our discussion of the B+-tree methods.

- We let “ $\langle \rangle$ ” denote an empty sequence, and let “ $+$ ” denote the concatenation operator when applied to a sequence.
- For leaf node x , and index $0 \leq i \leq x.size()-1$, we define

$$seq(x, i) = \begin{cases} \langle \rangle, & \text{if } x = AFT \\ x.data[i] + seq(x.next, 0) & i = x.size()-1 \\ x.data[i] + seq(x, i+1) & \text{otherwise.} \end{cases}$$

Abstraction Function: Let T be a B+-tree. The abstraction function is

$$AF(T) = seq(FORE.next, 0).$$

Design Notes: See the discussion in Chapter 36.

Optimizations: There is no need for each leaf node to have the array of children pointers. A more space efficient implementation would define a node interface, that is implemented by both an internal node and a leaf node. This optimization would allow the leaf node to not have any children references. See the implementation of tries (Chapter 39) for an illustration of an implementation of a tree-based data structure in which the leaf nodes are of a different type than the internal nodes.

Another option to reduce the space usage would be to use the leftmost and rightmost children references in the leaf nodes in place of *prev* and *next*. However, for ease of exposition, our implementation of the B+-tree leaf node just extends the B-tree node.

37.3 Representation Properties

We inherit all the B-tree properties and also add the following properties. Since all elements in a B+-tree are held in a leaf node, together with INORDER, it follows that $AF(T)$ is the sequence of elements in sorted order.

ELEMENTATLEAF: If e is an element in this collection, then some reachable leaf node must contain e .

SORTEDLEAFCHAIN: Let x be any leaf node reachable from the root. All elements in x are at least as large as all elements in leaf node $x.prev$. Also, for any leaf node other than AFT , $x.next.prev = x$.

INTERNALNODEELEMENTS: For any internal node x reachable from the root, if for index $i \in \{0, 1, \dots, x.size()-1\}$, element $e = x.data[i]$ is in the collection, then e must be in some leaf node in $T(x.child[i])$.

37.4 Leaf Node Inner Class

BTreeNode

└ *LeafNode*

In this section, we describe the *LeafNode* inner class that extends the *BTreeNode* class by introducing *prev* and *next*, which form a sorted doubly linked list over the leaf nodes.

```
public class LeafNode extends BTreeNode {
    LeafNode prev;
    LeafNode next;
```

The *setNext* method takes *ptr*, a reference to the leaf node to place after this leaf node in the leaf chain.

```
final void setNext(LeafNode ptr) {
    next = ptr;
    ptr.prev = this;
}
```

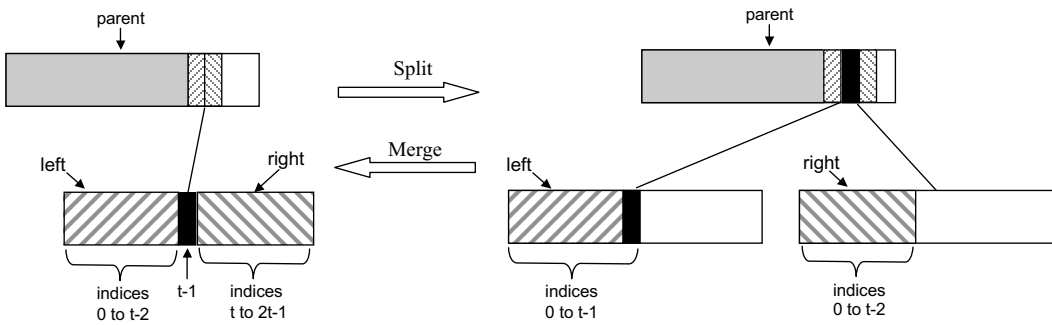
The *split* method splits a leaf node, and returns a reference to the newly created node. It requires that this node is full. The differences from the *BTreeNode* split method are (1) the median element is not removed from the left child, but rather a copy is added to the parent, (2) the t^{th} child of this node must be reset to *FRONTIER* since the inherited *move* sets it to *null*, and (3) the new leaf node is added into the sorted leaf chain. See [Figure 37.2](#) for an illustration.

```
LeafNode split(){
    LeafNode left = this;
    LeafNode right = new LeafNode();
    move(left, t, right, 0, t-1);
    children[t] = (BTreeNode) FRONTIER;
    E copyUp = data.get(t-1);
    addToParent(copyUp, right);
    right.setNext(left.next);
    left.setNext(right);
    return right;
}
```

//create new right child
//move last t-1 elements to new leaf
//set child t to FRONTIER
//median element, to copy up (don't remove)
//preserve SortedLeafChain

Correctness Highlights: By the correctness of *addToParent*, the elements in the parent are maintained in sorted order, and *PARENT* is preserved. Since the elements that remain in the *left* all precede *moveUp*, and the elements moved to the right are all at least as large as *moveUp* (and are kept in the same order), *INORDER* is maintained. Since all elements that were in this node (i.e., in *left*) either remain in *left* or are moved to *right*, so *ELEMENTATLEAF* is preserved.

The only violation to *FRONTIER* after *move* executes is that child *t* of this leaf node is set to *null*, but the median remains as *data[t-1]*. Thus setting *child[t]* to *FRONTIER*, preserves *FRONTIER*. Either the root splits, in which case the height grows for all paths, or the height remains unchanged. Thus *BALANCED* is maintained. The split node has *t* elements, and *t + 1* children, and the newly created node has *t - 1* elements and *t* children, thus *NODEUTILIZATION* is maintained. Since all elements that were reachable before, remain reachable, *REACHABLE* is preserved. Finally, *right* is added to the leaf chain after *left*, so *SORTEDLEAFCHAIN* is preserved. ■

**Figure 37.2**

An illustration of the B+-tree split and merge methods.

The *merge* method takes *parent*, the parent of the two nodes that are to be merged, and *index*, the index of this node in its parent's *children* array. It merges this node with its neighboring sibling to the right. This method requires that this node and its right sibling are minimum sized. The changes from the B-tree *merge* method are (1) the element *e* in the parent for which this node is the left child is not moved into the merged node, and (2) the right sibling is removed from the sorted leaf chain.

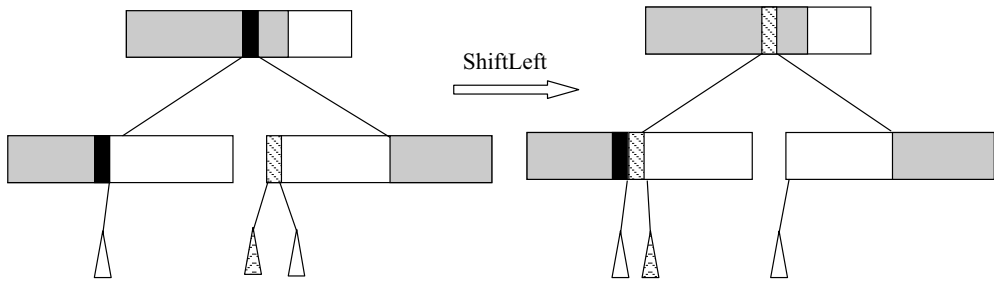
```
void merge(BTreeNode parent, int index) {
    LeafNode rightSibling = (LeafNode) parent.child(index+1);
    parent.remove(index);
    move(rightSibling, 0, this, t-1, t-1);
    setNext(rightSibling.next);                                //preserve SortedLeafChain
    if (root.size() == 0){
        root = root.child(0);                                  //preserve InOrder and Reachable
        ((BTreeNode) root).parent = null;                       //preserve Parent
    }
}
```

Correctness Highlights: By the correctness of the SortedArray *remove* methods, the elements in the parent are maintained in sorted order. Likewise, the elements in the node being merged are both in sorted order. By INORDER, $T(\text{left}) \leq \text{parent.data}(\text{index}) \leq T(\text{right})$, thus the elements are placed in *left* in sorted order, maintaining INORDER. If *parent* holds just one element, then the height decreases by 1 for all paths. Otherwise, the height remains unchanged. Thus, BALANCED is preserved. By the requirement of the *merge* method, *left* and *right* have $t - 1$ elements. So after the *merge*, *left* will have $2t - 2$ elements. Thus, NODEUTILIZATION is maintained.

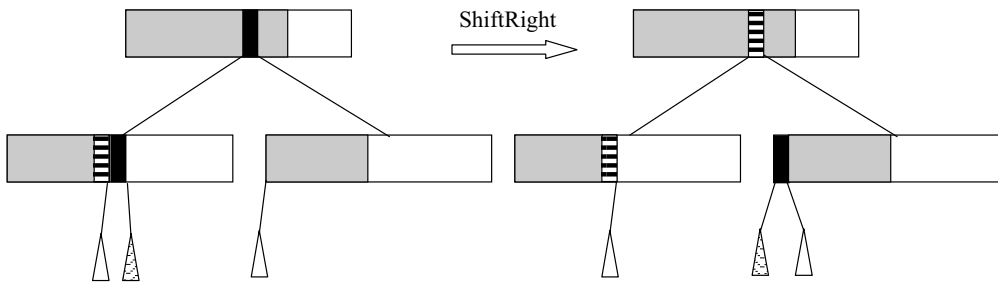
Since all elements in *rightSibling* are moved into this node, ELEMENTATLEAF is preserved. Removing *rightSibling* from the leaf chain preserves SORTEDLEAFCHAIN. Finally, INTERNAL-NODEELEMENTS can be violated only when an element is added to an internal node, and that does not occur here.

Finally, if the size of the root has been reduced to zero, then the only child of the current root (that at index 0) becomes the new root. ■

The *shiftLeft* method takes *parent*, the parent of this node, and *i*, the index of this node in its parent's *children* array. It requires that the right sibling of this node is not minimum sized, and that this node is not full. Figure 37.3 illustrates the changes. The only difference from the B-tree *shiftLeft* method is that the leftmost element of the right child, must not only be placed in the parent,

**Figure 37.3**

The B+-tree shift left method. The unshaded portions of each node are unused. The shaded child is the only one which is moved. All other children are unchanged. For illustrative purposes just two of the unchanged children are shown.

**Figure 37.4**

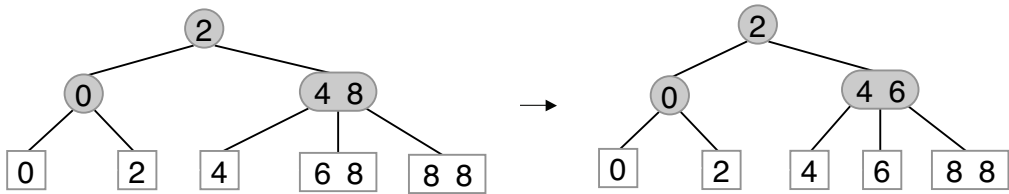
This diagram illustrates the B+-tree shift right method. The unshaded portions of the nodes are unused. The shaded child is the only one which is moved. All other children are unchanged. For illustrative purposes just two of the unchanged children are shown.

but also must be the element moved into this node. By copying it into the position of the element of the parent that will be moved to its left child, this goal is achieved.

```
void shiftLeft(BTreeNode parent, int i) {
    parent.data.set(i, parent.child(i+1).data.get(0));
    super.shiftLeft(parent, i);
}
```

Correctness Highlights: Moving element 0 of the right child into position i of the parent, prior to calling the superclass *shiftLeft* method, preserves both ELEMENTATLEAF and INTERNALNODEELEMENTS. No leaf nodes are removed or rearranged, so SORTEDLEAFCHAIN is preserved. All other properties are preserved by the B-tree class *shiftLeft* method. ■

The *shiftRight* method takes *parent*, the parent of this node, and i , the index of this node within the parent. The index i identifies the element of the parent whose left child's rightmost element should be shifted into this node, which is the right child of element i . This method requires that the left child is not at its minimum size, and that the right child is not at its maximum size. Figure 37.4 illustrates the changes. There are two key differences from the B-tree *shiftRight* method. First, analogous to *shiftLeft*, the rightmost element of the left child, must not only be placed in the parent, but also must be the element moved into this node. By copying it into position $i - 1$ of the parent before calling the inherited *shiftRight* method, this goal is achieved. Second, to preserve INTERNALNODEELEMENTS, the second largest element in the left sibling must be copied into the parent at $data[i-1]$.

**Figure 37.5**

B+ tree example in which the value of an element in an internal node must be updated to preserve INTERNALNODEELEMENTS. In this example, 8 is removed from leaf holding 6 and 8 from the tree on the left to obtain the tree on the right. In general, this situation could occur at a higher level of the B-tree, versus at the parent of the leaf node for which the element is being extracted.

```

void shiftRight(BTreeNode parent, int i) {
    BTreeNode left = parent.child(i-1);
    parent.data.set(i-1, left.data.get(left.size()-1));
    super.shiftRight(parent, i);
    LeafNode leftSibling = (LeafNode) parent.child(i-1);
    parent.data.set(i-1, leftSibling.data.get(leftSibling.size()-1));
}

```

Correctness Highlights: Moving the last element of the left child into element $i - 1$ of the parent, prior to calling the superclass *shiftRight* method, preserves ELEMENTATLEAF. No leaf nodes are removed or rearranged, so SORTEDLEAFCHAIN is preserved. In order to preserve INTERNALNODEELEMENTS, after the superclass *shiftRight* method executes, the last element of the left child must be copied to element $i - 1$ of the parent. All other properties are preserved by the B-tree class *shiftRight* method. ■

Recall that the *extract* method takes i , the index of the element to be removed. It requires that this node is not minimum sized. The change required in the B+ tree from the B-tree is to preserve INTERNALNODEELEMENTS while also preserving INORDER. For example, consider the situation shown at the left of Figure 37.5 in which the 8 is extracted from the leaf node x that holds 6 and 8. (So $x.extract(1)$ is called.) Since x is not minimum sized, removing it preserves all of the B-tree properties. However, if no other changes are made to the B+ tree then INTERNALNODEELEMENTS is violated for the element 8 at the internal node holding 4 and 8, because the greatest element in its left subtree is no longer an 8. First, observe that this situation can occur only when the element being removed is the rightmost element in x , and its predecessor is smaller than it. To avoid this potential violation, the *extract* method follows the parent references back up the search path to find the deepest internal node y (if any), for which the element e_x being removed is in the left subtree of an element e_y in node y that is equivalent to e_x . When this occurs e_y is replaced in node y by the predecessor of e_x . The node y can be anywhere on the search path to the root, including possibly the root. The right side of Figure 37.5 shows the B+ tree that results when the 8 that was in the node holding 6 and 8 is removed.

```

void extract(int i) {
    E removed = data(i);
    super.extract(i);
    if (this != root && i == size()) {           //removing the rightmost element in this node

```

```

    E pred = data(i-1);
    if (comp.compare(pred, removed) < 0) //violation only occurs when pred different
        for (BTreeNode ptr = this; ptr != root; ptr = ptr.parent)
            if (ptr.pIndex < ptr.parent.size() &&
                comp.compare(ptr.parent.data(ptr.pIndex), removed) == 0) {
                ptr.parent.data.set(ptr.pIndex, pred);
                return;
            }
    }
}

```

Correctness Highlights: By the correctness of the inherited *extract* method, the desired element (already stored in *removed*) is removed from the node. If this node is the root, then no further modifications are required since there are no other nodes in this B+-tree.

We now argue that if the element *e* removed was not the rightmost element (i.e., $i \neq \text{size}()$) then no further changes are needed. The only property that could have a violation is INTERNALNODEELEMENTS. We use a proof by contradiction to argue that no violation can occur when *e* is not the rightmost element. Suppose, for contradiction, that a violation occurs when *e* is not the rightmost element in its node. Let node *y* be a deepest node in which a violation occurs, and let $e_j = y.\text{data}[j]$ for which e_j is in the collection but not in $T_j = T(y.\text{child}[j])$. Since INTERNALNODEELEMENTS held prior to removing *e*, e_j must have been in T_j . Then, at that time, by INORDER, e_j must have been the rightmost element in T_j . By assumption, no rightmost element is changed, so this violation could not have occurred.

Next we consider when the rightmost element of this node is the one being removed. If the predecessor of *e* is equivalent to *e* then the value of the rightmost element of this node does not change. So by the argument given above INTERNALNODEELEMENTS cannot be violated. So we focus on the case when the predecessor of *e* is less than *e*. The for loop moves *ptr* up the tree until it reaches the root. The element that could cause a violation of INTERNALNODEELEMENTS is the element e_j at index *j* of the parent, for which e_j is in the collection but not in $T(\text{ptr.parent.child}[j])$. Thus, if *ptr* is the rightmost child of its parent, then no violation could occur at this level. By the correctness of the *child* method, *ptr.pIndex* is the index for which $T(\text{ptr}) = T(\text{ptr.parent.child}[\text{pIndex}])$ so the only possible violation could be caused with respect to e_j since $T(\text{ptr})$ is the only subtree of *parent* that has changed.

Let *y* be a deepest node (if any) for which $y.\text{data}[j]$ is equivalent to *e*. In this case, $y.\text{data}[j]$ is replaced by the predecessor of *e*. We argue that INTERNALNODEELEMENTS is restored by this change. As discussed earlier, a violation could only occur if *e* is the rightmost element in $T(\text{ptr})$. By INORDER this implies that all elements that remain in $T(\text{ptr})$ are no larger than *pred*. Thus, replacing $y.\text{data}[j]$ by *pred* restores INTERNALNODEELEMENTS while preserving INORDER. No violations of INTERNALNODEELEMENTS can occur further up the tree, since by INTERNALNODEELEMENTS with respect to *y.parent*, if $y.\text{data}[j]$ occurs, then it must occur in $T(y)$ and that still holds. Thus, *extract* can return, having preserved the property.

If *y* does not exist, then *ptr* continues up the tree until all nodes (including the root) are examined for a possible violation. ■

37.5 B+-Tree Methods

In this section we present internal and public methods for the BPlusTree class.

37.5.1 Constructors and Factory Methods

The constructor that takes no parameters creates an empty B+-tree with $t = 2$ that uses the default comparator.

```
public BPlusTree() {
    this(Objects.DEFAULT_COMPARATOR, 2);
}
```

The constructor that takes a single parameter t , the order for the B+-tree, creates an empty B+-tree of order t .

```
public BPlusTree(int t) {
    this(Objects.DEFAULT_COMPARATOR, t);
}
```

The constructor that takes *comp*, the function used to compare two elements, and t , the order of the B+-tree, creates an empty B+-tree of order t that uses the given comparator.

```
public BPlusTree(Comparator<? super E> comp, int t) {
    super(comp, t);
    FORE = new LeafNode();
    AFT = new LeafNode();
    ((LeafNode) FORE).setNext((LeafNode) AFT);
}
```

Correctness Highlights: This method enforces ORDEREDLEAFCHAIN. All other properties are satisfied by the superclass constructor. ■

The factory method *createRoot* creates and initializes a new empty root node.

```
protected void createRoot(){
    root = new LeafNode();
    ((LeafNode) root).setNext((LeafNode) AFT);
    ((LeafNode) FORE).setNext((LeafNode) root);
}
```

Correctness Highlights: This method preserves ORDEREDLEAFCHAIN. ■

37.5.2 Representation Accessors

The *leftmost* method returns the leftmost node in the B+-tree. The ordered leaf chain enables us to provide a more efficient implementation of this method than that in the abstract search tree class.

```
TreeNode leftmost() {
    return ((LeafNode) FORE).next;
}
```


Correctness Highlights: Follows from ORDEREDLEAFCHAIN. ■

The *rightmost* method returns the rightmost node in the B+-tree. Again, the ordered leaf chain enables us to provide a more efficient implementation of this method than that in the abstract search tree class.

```
TreeNode rightmost() {
    return ((LeafNode) AFT).prev;
}
```

Correctness Highlights: Follows from ORDEREDLEAFCHAIN. ■

37.5.3 Algorithmic Accessors

The *find* method takes *target*, the target element. It returns a reference to the leaf node where the search ends (which would be *FRONTIER* when there is no equivalent element in the collection. If there is no equivalent element the collection, then *find* returns the frontier node where the target would be inserted with the *parent* field set to the node that preceded it on the search path. This method sets the global instance variable *curIndex* to hold the position for an occurrence of the target, if any, or otherwise the position where it would be inserted in the node. Unlike the B-tree *find* method that can stop at an internal node if the desired element is found, since a B+-tree only holds the elements in the collection at the leaf nodes, the B+-tree *find* method must continue the search until reaching a leaf.

```
protected BTreeNode find(E target) {
    BTreeNode ptr = (BTreeNode) root;                                //start at the root
    while (!ptr.isFrontier()) {                                       //until a frontier node is reached
        curIndex = ptr.data.find(target);                            //look for the target in the node
        if (curIndex < ptr.size() && ptr.isLeaf() &&
            comp.compare(target, ptr.data.get(curIndex)) == 0)
            return ptr;                                             //return the ptr to the current node
        ((BTreeNode) FRONTIER).parent = ptr;                       //set frontiers' parent to ptr
        ptr = ptr.child(curIndex);                                  //go to the appropriate child
    }
    return ptr;                                                     //not found, return ptr to frontier node with parent set
}
```

Correctness Highlights: We argue inductively that if there is an element equivalent to the target in this collection, then it occurs at some leaf node in $T(ptr)$. By REACHABLE and ELEMENT-ATLEAF, this invariant holds before entering the while loop. For the inductive step we consider the following two cases:

The target is held in *ptr*. By the correctness of the sorted array *find* method, *curIndex* gives the index of some occurrence of the target in *ptr*. By INTERNALNODEELEMENTS, if the target is in the collection then some occurrence of it occurs in a leaf node in $T(node.child(curIndex))$.

The target is not in *ptr*. By the correctness of *find*, *curIndex* is the insert position for the target in *ptr*. By INORDER it follows that if the target is in the collection then some occurrence of it occurs in a leaf node in $T(\text{node.child}(\text{curIndex}))$.

By SORTEDLEAFCHAIN if the element is in the collection, then it will appear in a leaf node. Combined with the above cases, it follows that this invariant is preserved by the while loop.

Setting the parent of *FRONTIER* at each step of the loop ensures, that it will reference the predecessor on the search path if the target is not in the collection. By *FRONTIER*, the loop will terminate. Finally, by INUSE we are guaranteed that no comparisons made involves a frontier node. ■

The *findFirstInsertPosition* method takes *target*, the target element. It returns a reference to the B-tree node where the target would be inserted to precede any equivalent elements in the iteration order. It sets the global variable *curIndex* to hold the insert position for the target in the returned node.

```
BTreeNode findFirstInsertPosition(E target) {
    BTreeNode ptr = (BTreeNode) root;                                //start at the root
    while (!ptr.isFrontier()) {                                       //until a frontier node is reached
        curIndex = ptr.data.findFirstInsertPosition(0, ptr.size()-1, target);
        if (curIndex < ptr.size() && ptr.isLeaf() &&
            comp.compare(target, ptr.data.get(curIndex)) == 0)
            return ptr;
        ((BTreeNode) FRONTIER).parent = ptr;                       //set frontiers' parent to ptr
        ptr = ptr.child(curIndex);                                   //go to the appropriate child
    }
    return ptr.parent;                                              //not found, return ptr to frontier node with parent set
}
```

Correctness Highlights: Like that of *find*, except that it is modified to use the SortedArray *findFirstInsertPosition* method versus the *find* method. When the while loop terminates, *returnNode* is *null* if and only if there is no equivalent element in the target. In this case, the search path would have continued at *children[curIndex]* of *FRONTIER.parent*. By INORDER, it follows that inserting the target at index *curIndex* of *FRONTIER.parent* would place it in the iteration order in sorted order.

When *returnNode* is not *null*, it is known that the first occurrence of the target in the iteration order is at *returnNode.data[curIndex]*. Observe that inserting the target in this position would place it in the iteration order just before any equivalent elements. ■

The *findLastInsertPosition* method takes *target*, the target element. It returns a reference to the B-tree node where the target would be inserted to follow any equivalent elements in the iteration order. It sets the global variable *curIndex* to hold the insert position for the target in the returned node.

```
BTreeNode findLastInsertPosition(E target) {
    BTreeNode ptr = (BTreeNode) root;                                //start at the root
    while (!ptr.isFrontier()) {                                       //until a frontier node is reached
        curIndex = ptr.data.findLastInsertPosition(0, ptr.size()-1, target);
        if (curIndex < ptr.size() && ptr.isLeaf() &&
            comp.compare(target, ptr.data.get(curIndex)) == 0)
            return ptr;
    }
```

```

        ((BTreeNode) FRONTIER).parent = ptr;           //set frontiers' parent to ptr
        ptr = ptr.child(curIndex);                     //go to the appropriate child
    }
    return ptr.parent;                                //not found, return ptr to frontier node with parent set
}

```

Correctness Highlights: Like *findFirstInsertPosition* except that the SortedArray *findLastInsertPosition* method is used. Also, if the element is found, then the right (versus left) child of that element is followed. ■

The *pred* method takes *x*, a reference to a leaf node holding the target element, and *index*, the index of the target element. It returns a reference to the leaf node holding the predecessor of the target element, or FORE if it has no predecessor. This method sets the global variable *curIndex* to hold the index of the predecessor of the target element, if it exists. The existence of the sorted leaf chain greatly simplifies this method as compared to that of the B-tree.

```

TreeNode pred(BTreeNode x, int index) {
    if (index > 0)
        curIndex = index-1;
    else {
        x = ((LeafNode) x).prev;
        curIndex = x.size()-1;
    }
    return x;
}

```

Correctness Highlights: By INORDER if the element is not the leftmost element in leaf node *x*, then the element immediately before it in the same leaf node is the predecessor. Otherwise, by SORTEDLEAFCHAIN and ELEMENTATLEAF, the last element in the leaf node that precedes *x* in the sorted leaf chain is the predecessor. ■

The *succ* method takes *x*, the BTreeNode holding the element for which the successor in the iteration order is to be found, and *index*, the index of the element in *x*. It returns a reference to the BTreeNode holding the successor or AFT if there is no successor. In addition, this method has the side effect of setting *curIndex* to the index of the successor in the BTreeNode returned.

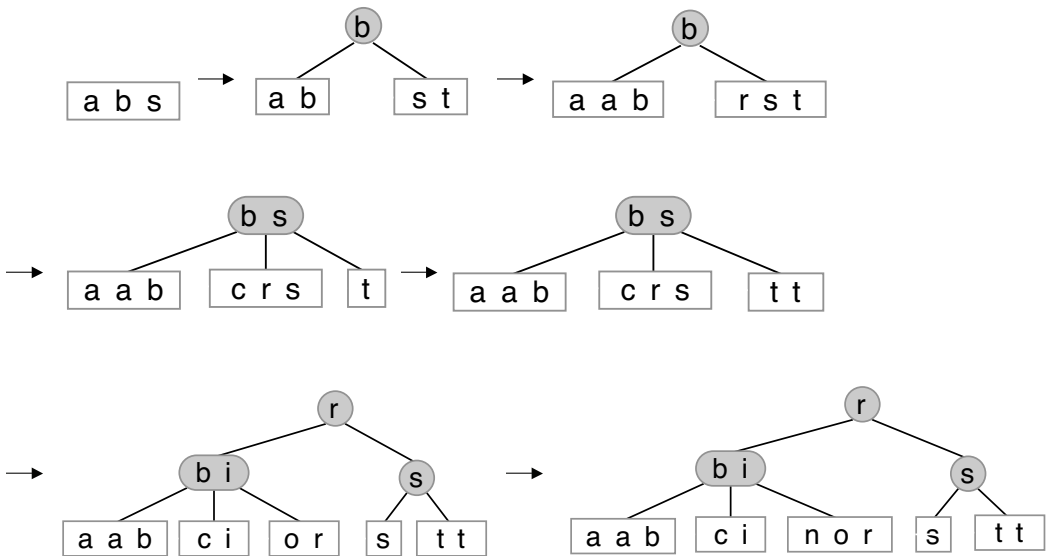
```

TreeNode succ(BTreeNode x, int index){
    if (index < x.size()-1)
        curIndex = index+1;
    else {
        x = ((LeafNode) x).next;
        curIndex = 0;
    }
    return x;
}

```

Correctness Highlights: The correctness mirrors that of the *pred* method. ■

The ordered leaf chain enables more efficient iteration using an iterator, as compared to using an inorder traversal. Thus both the *traverseForVisitor* and *writeElements* method are replaced to use

**Figure 37.6**

B+-tree insertion example with $t = 2$, obtained by inserting the letters in “abstraction,” in that order.

an iterator as in `AbstractCollection`, instead of using the inherited `inorder traversal` method from `AbstractSearchTree`. The `traverseForVisitor` method takes v , a visitor. It traverses the collection applying v to each element.

```
protected void traverseForVisitor(Visitor<? super E> v) throws Exception {
    for (E e : this)
        v.visit(e);
}
```

The `writeElements` method takes `sb`, the string builder to fill with a comma-separated string of the elements in the collection in sorted order.

```
protected void writeElements(StringBuilder sb) {
    Locator<E> loc = iterator();
    while (loc.advance()) {
        sb.append(loc.get());
        if (loc.hasNext())
            sb.append(", ");
    }
}
```

37.5.4 Content Mutators

All of the changes required for insertion occur within the `split` method of the leaf node. [Figure 37.6](#) illustrates the construction of a B+-tree holding the letters of “abstraction,” inserted in that order.

We now describe the `remove` method that takes x , the tree node holding the element to remove, and $index$, the index of the element to remove. The node x is guaranteed to be a leaf.

```

void remove(BTreeNode x, int index) {
    super.remove(x, index);
    if (isEmpty()) //remove empty node that was the root
        ((LeafNode) FORE).setNext((LeafNode) AFT);
}

```

Correctness Highlights: By the correctness of the *shiftLeft*, *shiftRight*, and *extract* methods used by the B-tree node *remove* method, all properties are preserved with one small exception. After removing the last element from the collection, the sorted leaf chain includes an empty leaf node that used to be the singleton root node. The final line restores SORTEDLEAFCHAIN in this situation by having AFT directly follow FORE in the leaf chain. ■

The *clear* method removes all elements from the collection.

```

public void clear(){
    super.clear();
    ((LeafNode) FORE).setNext((LeafNode) AFT);
}

```

Correctness Highlights: SORTEDLEAFCHAIN is restored by having AFT immediately follow FORE in the leaf chain. The rest follows from the inherited method. ■

37.6 Performance Analysis

The asymptotic time complexity of each public method of the BPlusTree class is shown in [Table 37.7](#), and the asymptotic time complexity for each public method of the Marker class is shown in [Table 37.8](#).

The analysis is like the B-tree analysis with the following two exceptions. The height of a B+-tree is slightly greater since all the elements are held in leaf nodes. More specifically, in a B+-tree roughly $1 - \frac{1}{t}$ of the nodes are leaf nodes. So the space usage increases by roughly a multiplicative factor of $t/(t-1)$. When $t = 2$ the space usage is almost twice as large. However, for larger values of t there is a relatively small space overhead.

The height of both a B-tree and B+-tree is at most $\log_t((n' + 1)/2) \approx \log_t n'$ where n' is the total number of nodes. So when $t = 2$ the height of a B+-tree will be one more than the height of a B-tree holding the same elements. For large values of t , their expected height is basically the same. This means that there is almost no extra computational cost to locate, insert, or remove an element caused by storing all elements in the leaves.

The second major difference in terms of the time complexity between a B-tree and B+-tree is that placing all elements in the leaves in a threaded chain means that moving forward or backward in the iteration order takes constant time for a B+-tree, as compared to logarithmic time for the B-tree. Thus, the *advance*, *retreat*, *next*, and *hasNext* methods all take constant time and require reading no more than one disk page to possibly retrieve the neighboring leaf node. Both *predecessor* and *successor* take constant time once the search for the desired element has been performed, so they are more efficient than for a B-tree. However, the asymptotic costs are not changed.

The *toString* method can traverse at the leaf level instead of using the inorder traversal used by the B-tree. However, the asymptotic complexity of the *toString* method is not changed.

The remainder of the time complexity analysis for a B+-tree is just like that for the B-tree.

method	time complexity	disk pages read
constructors	$O(1)$	0
<i>ensureCapacity(x)</i>	$O(1)$	0
<i>iterator()</i>	$O(1)$	0
<i>iteratorAtEnd()</i>	$O(1)$	0
<i>trimToSize()</i>	$O(1)$	0
<i>max()</i>	$O(1)$	1
<i>min()</i>	$O(1)$	1
<i>predecessor(o)</i>	$O(1)$	1
<i>successor(o)</i>	$O(1)$	1
<i>contains(o)</i>	$O(\log_2 n)$	$\log_t n$
<i>getLocator(o)</i>	$O(\log_2 n)$	$\log_t n$
<i>getEquivalentElement(o)</i>	$O(\log_2 n)$	$\log_t n$
<i>add(o)</i>	$O(t \log_t n)$	$2 \log_t n$
<i>remove(o)</i>	$O(t \log_t n)$	$2 \log_t n$
<i>accept(v)</i>	$O(n)$	$n/(t - 1)$
<i>toArray()</i>	$O(n)$	$n/(t - 1)$
<i>toString()</i>	$O(n)$	$n/(t - 1)$
<i>retainAll(c)</i>	$O(n c + n \log_t n)$	$2 c \log_t n$
<i>addAll(c)</i>	$O(c \log_t (n + c))$	$2 c \log_t (n + c)$

Table 37.7 Summary of the asymptotic time complexities for the collection public methods when using the B+ Tree data structure to implement the OrderedCollection ADT.

locator method	time complexity	disk pages read
constructor	$O(1)$	0
<i>get()</i>	$O(1)$	1
<i>advance()</i>	$O(\log_2 n)$	$\log_t n$
<i>hasNext()</i>	$O(\log_2 n)$	$\log_t n$
<i>next()</i>	$O(\log_2 n)$	$\log_t n$
<i>retreat()</i>	$O(\log_2 n)$	$\log_t n$

Table 37.8 Summary of the time complexities for the public methods of the BPlusTree Marker class.

37.7 Quick Method Reference

BPlusTree Public Methods

- p. 584 *BPlusTree*()
- p. 584 *BPlusTree*(Comparator<? super E> comp, int t)
- p. 584 *BPlusTree*(int t)
- p. 98 void **accept**(Visitor<? super E> v)
- p. 479 void **add**(E element)
- p. 100 void **addAll**(Collection<? extends E> c)
- p. 100 void **clear**()
- p. 97 boolean **contains**(E value)
- p. 99 void **ensureCapacity**(int capacity)
- p. 476 E **get**(int r)
- p. 96 int **getCapacity**()
- p. 96 Comparator<? super E> **getComparator**()
- p. 98 E **getEquivalentElement** (E target)
- p. 568 Locator<E> **getLocator**(E x)
- p. 96 int **getSize**()
- p. 96 boolean **isEmpty**()
- p. 567 Locator<E> **iterator**()
- p. 567 Locator<E> **iteratorAtEnd**()
- p. 478 E **max**()
- p. 477 E **min**()
- p. 561 E **predecessor**(E target)
- p. 479 boolean **remove**(E element)
- p. 100 void **retainAll**(Collection<E> c)
- p. 562 E **successor**(E target)
- p. 97 Object[] **toArray**()
- p. 97 E[] **toArray**(E[] array)
- p. 98 String **toString**()
- p. 99 void **trimToSize**()

BPlusTree Internal Methods

- p. 97 int **compare**(E e1, E e2)
- p. 558 void **createRoot**()
- p. 97 boolean **equivalent**(E e1, E e2)
- p. 558 BTreeNode **find**(E target)
- p. 558 BTreeNode **findFirstInsertPosition**(E target)
- p. 559 BTreeNode **findLastInsertPosition**(E target)
- p. 476 int **getLastNodeSearchIndex**()
- p. 563 BTreeNode **insert**(E element)
- p. 477 BTreeNode **leftmost**()
- p. 476 BTreeNode **leftmost**(BTreeNode x)
- p. 562 void **move**(BTreeNode from, int fromIndex, BTreeNode to, int toIndex, int num)
- p. 560 BTreeNode **pred**(BTreeNode x, int index)
- p. 565 void **remove**(BTreeNode x, int index)
- p. 567 void **remove**(BTreeNode x)
- p. 477 BTreeNode **rightmost**()
- p. 477 BTreeNode **rightmost**(BTreeNode x)
- p. 561 BTreeNode **succ**(BTreeNode x, int index)
- p. 99 void **traverseForVisitor**(Visitor<? super E> v)
- p. 478 void **traverseForVisitor**(Visitor<? super E> v, BTreeNode x)
- p. 98 void **writeElements**(StringBuilder s)
- p. 479 void **writeElements**(StringBuilder sb)

BPlusTree.LeafNode Internal Methods

- p. 582** void **extract**(int i)
- p. 580** void **merge**(BTreeNode parent, int index)
- p. 579** void **setNext**(LeafNode ptr)
- p. 581** void **shiftLeft**(BTreeNode parent, int i)
- p. 581** void **shiftRight**(BTreeNode parent, int i)
- p. 579** LeafNode **split**()

Chapter 38

Skip List Data Structure

AbstractCollection<E> implements *Collection*<E>

↑ **SkipList**<E> implements *OrderedCollection*<E>, *Tracked*<E>

Uses: Java array and references

Used By: TaggedSkipList (Section 49.9.8), historical event collection case study (Sections 29.1 and 50.1)

Strengths: Maintains expected logarithmic search cost without any need to restructure. Included within a skip list is a sorted doubly linked list holding all elements in the collection enabling very fast iteration. Removing an element that has been located (either by a search or via a locator) takes expected constant time. A skip list is particularly well suited for a one-dimensional range search. To perform a range search on a skip list, position a tracker at the predecessor of the low end of the range, and then iterate over the collection until reaching an element beyond the desired range.

Weaknesses: While a skip list has expected logarithmic search time, the time to search for an element, or its insert position, is slower than for a balanced search tree. Also, since the storage is organized “vertically,” cache line misses are typically more frequent for a skiplist than for a search tree.

Critical Mutators: none

Competing Data Structures: A sorted array (Chapter 30) is preferable if fast search time is very important and elements added or removed are near the maximum, or when there are no mutations after inserting all elements.

If mutations occur, yet the majority of the methods involve locating an element (via a search), then a red-black tree (Chapter 34) is preferable since the search time is generally faster unless the elements being added or removed are near the front of the iteration order.

If it is important that recently accessed elements can be accessed quickly then a splay tree (Chapter 35) is a good option, provided that it is acceptable for some operations to take linear (but amortized logarithmic) time.

For two-dimensional (or higher dimensional) range searches, either a k -d tree (Chapter 47) or quad tree (Chapter 48) should be considered.

38.1 Internal Representation

A skip list can be viewed as a sorted doubly linked list with additional structure to allow logarithmic time access to any position in the list, in a way reminiscent of the binary search algorithm (Section 30.3.3). More specifically, a skip list can be viewed as a series of sorted lists $L_0, L_1, \dots, L_h$

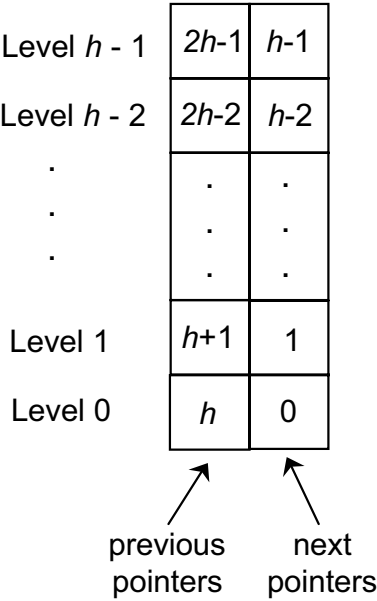


Figure 38.1
This diagram shows how we visually display the *links* array. Each box contains the index within *links* where that reference is held.

where all elements in the collection are in L_0 , and each element in L_i is also placed in L_{i+1} with probability $p = 1/4$. Each element in the collection has a reference to its predecessor and successor in each list L_i containing it. We define the **height** of an element at the number of lists in which it is contained. More specifically, if an element has height h then it is in lists $L_0, \dots, L_{h-1}$.

A skip list is composed of **tower** objects, where each tower holds a reference to its element, and an array called *links* containing the next and previous references for each L_i that contains the element.

Instance Variables and Constants: The following instance variables and constants are defined for the SkipList class. Also, *size*, *comp*, and *version* are inherited from AbstractCollection.

The tower *head* is a sentinel head tower that serves the role of FORE. Similarly, the tower *tail* is a sentinel tail tower that serves the role of AFT. The constant *DEFAULT_HEIGHT* gives the default value for the initial height of *head* and *tail*. The constant *MAX_HEIGHT* is the maximum height allowed for any tower. The integer *height* holds the **current height** for the skip list, which is defined as the greatest height among all towers in the skip list. Finally, *randseq* is a random number generator used for randomly selecting the height for each tower.

```
public static final int DEFAULT_HEIGHT = 2;
public static final int MAX_HEIGHT = 32;
int height;
Tower<E> head;
Tower<E> tail;
java.util.Random randseq;
```

//max height of any tower
//sentinel head, also serves role of FORE
//sentinel tail, also serves role of AFT
//for random number generation

Populated Example: Figure 38.2 shows a populated example of a skip list tree holding the letters of “abstract.” An abstract view of the skip list of Figure 38.2 is shown in Figure 38.3.

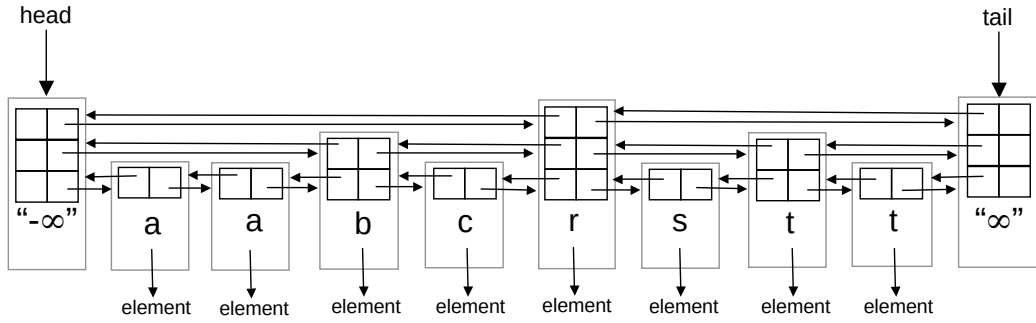


Figure 38.2

A populated example for a skip list holding the letters of “abstract.” This figure shows the internal representation. Note that the “b” and the first “t” are in lists L_0 and L_1 . The “r” is in lists L_0 , L_1 , and L_2 . All other elements are in only L_0 . An abstract view of this skip list is shown in Figure 38.3.

$$\begin{array}{rcl}
 L_2: & & r \\
 L_1: & b & r \quad t \\
 L_0: & a & a \quad b \quad c \quad r \quad s \quad t \quad t
 \end{array}$$

Figure 38.3

An abstract view of the skip list shown in Figure 38.2.

Terminology: We define the list L_i for $i = 0, \dots, \text{height} - 1$ as follows.

- For Tower x in list L_i , we let $x.\text{succ}(i)$ denote the next tower after x in L_i .
- We let $x_j^{(i)}$ denote the j^{th} tower in L_i where $x_{-1}^{(i)}$ is *head*. We define $|L_i|$ as the value of j for which $x_j^{(i)} = \text{tail}$, and define $x_{-1}^{(i)}, \dots, x_{|L_i|}^{(i)}$ recursively, as

$$x_j^{(i)} = \begin{cases} \text{head} & \text{for } j = -1 \\ x_{j-1}^{(i)}.\text{succ}(i) & \text{for } j = \{0, \dots, |L_i|.\} \end{cases}$$

- $L_i = \langle u_0^{(i)}, u_1, \dots, u_{|L_i|-1}^{(i)} \rangle$ such that $u_p^{(i)} = x_p^{(i)}.\text{data}$.
- We say that towers $x_{-1}^{(0)}, x_0^{(0)}, \dots, x_{n-1}^{(0)}$ are **reachable** since they can be accessed from *head*. These towers hold all the elements in the collection. We say that all other towers are **unreachable**. (Note that an unreachable tower might be accessible from a tracker, but it is not accessible by following the *next* references starting at *head*.)
- We say that L_0 is the **leaf level**.
- For all $i > 0$, L_i is an **index level**.

Abstraction Function: Let SL be a skip list. The abstraction function

$$AF(SL) = L_0.$$

Design Notes: To reduce the space overhead due to arrays, a single array of references, *links* is used to hold both the next and the previous references for each tower.

The value p is the probability with which a tower is selected to be part of list L_{i+1} given that it has already been selected to be part of list L_i . The provided implementation fixes $p = 1/4$. Another alternative would be to let $p = 1/2$. The expected search cost is the same when $p = 1/4$ and $p = 1/2$. When $p = 1/2$ the variance in the expected search cost is reduced. However, the expected height, and thus the space usage, is higher when $p = 1/2$. The optimal value for p is $1/e$ but the cost for the random number generation would be higher, and the expected cost savings are relatively insignificant when compared to $p = 1/4$.

The redirect chain makes use of both the **proxy design pattern**, and the idea of path compression that originated in the union-find data structure (Section 6.3).

Optimizations: If there is no need for *retreat* and *predecessor* to run efficiently (i.e., linear time would be acceptable), then the previous references are not needed which would enable the *links* array to be half the size. As done in the singly linked list (Chapter 15), this optimization would require that *remove* looks forward one. Observe, that other uses of the previous references (except in *retreat* and *predecessor*) are only to maintain them.

The cost of *contains* could be slightly improved by replacing the call to *findFirstOccurrence* with a call to a *find* methods that returns any occurrence of an element (if it is in the collection).

One could improve the efficiency of the method to get the element at rank r by maintaining in each tower the number of elements between it and the next tower of the same height. In that case, one would only drop to the next level when the desired rank is within the range of a tower, resulting in expected logarithmic time complexity.

38.2 Representation Properties

We inherit SIZE and introduce the following additional representation properties.

REACHABLE: The towers reachable from *head* (excluding *tail*) are exactly those holding an element in the collection.

SUBSET: All elements in the collection are in L_0 and for all $i > 0$, $L_{i+1} \subseteq L_i$. More specifically, each element in L_i is also placed in L_{i+1} with probability p where this decision is made independently for each element in L_i .

SORTED: For $i = 0, \dots, \text{height} - 1$, the elements of L_i are sorted in non-decreasing order. That is, for all i and $0 \leq p \leq n - 1$, $x_{p-1}^{(i)}.data \leq x_p^{(i)}.data$.

PREVIOUSREFERENCE: For any tower x (including *head*), let y be the tower that follows x in L_i . Then the previous link from y at level i references x . More formally, if $x.next(i) = y$, then $y.prev(i) = x$, for all x in the collection and for $x = \text{head}$.

HEIGHT: The instance variable *height* is the maximum value for *links.length/2* among all towers for elements in the collection. That is *height* is the number of levels of the tallest reachable tower, other than *tail*.

INUSE: For x any reachable tower, $x.pred(0) \neq \text{null}$.

REDIRECTCHAIN: When a tracked element is removed then the tracker is considered to be positioned just before the successor of the element at that time. For x any unreachable tower, the chain defined by the $links[0]$ references, starting at x , ends at the tower that x is considered to be just before. That is, the level 0 next references, starting at x , leads to the next element in the iteration order.

From SUBSET it follows that $|L_0| = size$ since all elements are in L_0 .

38.3 Tower Inner Class

Tower<*T*>

The skip list tower inner class has two instance variables: *element* holds a reference the element held in the tower, and *links* is the array holding references to the predecessor and successor in all lists containing *element*. For a tower of height h , $links[0]$, ..., $links[h-1]$ hold the next references for levels 0 to $h-1$, and $links[h]$, ..., $links[2h-1]$ hold the previous references for levels 0 to $h-1$. (See [Figure 38.1](#) for an illustration.) It is easily verified that the internal methods provided in the Tower class follow this convention.

```
T element;
Tower<T>[] links;
```

The constructor takes *element*, a reference to the element, and *towerHeight*, the desired height for the tower. It creates a *links* array of size $2 \cdot towerHeight$ so that there will be *towerHeight* next and previous references.

```
Tower(T element, int towerHeight) {
    this.element = element;
    links = (Tower<T>[]) new Tower[2 * towerHeight];
}
```

The *getTowerHeight* method returns the height of the tower. Since *links* is guaranteed to be exactly twice the tower height, by shifting *links.length* right by one bit, we obtain the correct height.

```
final int getTowerHeight() {
    return (links.length) >> 1;
}
```

The *prev* method takes *level*, the list to use, and returns the previous tower in L_{level} .

```
Tower<T> prev(int level) {
    return links[getTowerHeight()+level];
}
```

The *setPrev* method takes *level*, the list to use, and *t*, a reference for the tower that is to be placed before this tower in L_{level} .

```
void setPrev(int level, Tower<T> t) {
    links[getTowerHeight()+level] = t;
}
```

The *next* method takes *level*, the list to use, and returns the next tower in L_{level} .

```
Tower<T> next(int level) {
    return links[level];
}
```

The *setNext* method takes *level*, the list to use, and *t*, a reference for the tower that is to be placed after this tower in L_{level} .

```
void setNext(int level, Tower<T> t) {
    links[level] = t;
    t.setPrev(level, this);
}
```

Correctness Highlights: This method ensures that using *setNext* will maintain PREVIOUSREFERENCE since it always sets the previous reference so that if *this.next(level) = t* then *t.prev(level) = this*. ■

The *delete* method marks that a tower has been deleted from the collection by setting its level 0 predecessor to *null*.

```
void delete() {
    setPrev(0, null);
}
```

Correctness Highlights: This method preserves INUSE. ■

Similarly, the *isDeleted* method returns *true* exactly when an tower is marked as deleted.

```
boolean isDeleted() {
    return prev(0) == null;
}
```

Correctness Highlights: Follows from INUSE. ■

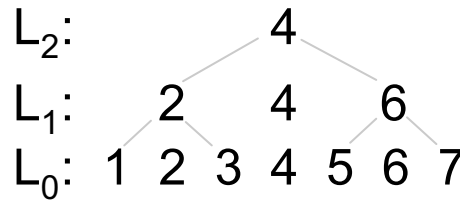
38.4 Skip List Methods

In this section, we present the internal and public methods for the SkipList class.

38.4.1 Constructors

The most general constructor takes two parameters *size*, the initial size for head and tail, and *comp*, the comparator that defines an ordering among the elements. It creates an empty skip list that uses the provided initial height for head and tail and the provided comparator.

```
public SkipList(int size, Comparator<? super E> comp) {
    super(comp);
    randseq = new java.util.Random();           //initialize random number sequence
}
```

**Figure 38.4**

An abstract view of a skip list holding $1, \dots, 7$ where exactly every other element in L_i also appears in L_{i+1} . In the top level of this skip list only one element remains. In gray we show the relationship between the skip list and balanced binary search tree with the same elements.

```

height = 0;                                     //no towers, so current height is 0
head = new Tower<E>(null, size);
tail = new Tower<E>(null, size);
head.setPrev(0, head);                          //satisfy InUse
for (int i = 0; i < size; i++)                  //in all lists, initially
    head.setNext(i, tail);                      //head points to tail
}

```

Correctness Highlights: The abstract collection constructor initializes *comp* to the provided comparator, and initializes *size* to 0 (satisfying SIZE). By initializing *height* to 0, HEIGHT is satisfied. The initial size of *head* and *tail* is initialized as specified. Setting the successor at each level in the head to be the tail, by definition causes $|L_i| = 0$ for all i , so SORTED and SUBSET hold. Setting the predecessor at each level in the tail to be the head satisfies PREVIOUSREFERENCE. INUSE is satisfied by initializing *head.pred(0)* to reference itself. Finally, REACHABLE, INUSE vacuously hold since no towers have been created for elements. ■

Several additional convenience constructors are provided to replace some parameters by the default values. Their correctness follows from that of the above constructor. The constructor that takes no arguments creates an empty skip list using the default height and comparator.

```

public SkipList() {
    this(DEFAULT_HEIGHT, Objects.DEFAULT_COMPARATOR);
}

```

The constructor that takes one parameter *size*, the initial size for head and tail, creates an empty skip list using the default comparator and the provided size for the height of the head and tail.

```

public SkipList(int size) {
    this(size, Objects.DEFAULT_COMPARATOR);
}

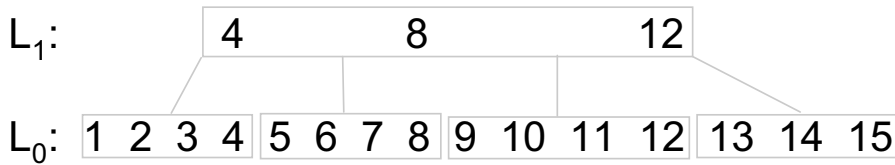
```

The constructor that takes *comp*, the comparator that defines an ordering among the elements, creates an empty skip list that uses the given comparator and the default height for head and tail.

```

public SkipList(Comparator<? super E> comp) {
    this(DEFAULT_HEIGHT, comp);
}

```

**Figure 38.5**

An abstract view of a skip list holding $1, \dots, 15$ where exactly every fourth element in L_i also appears in L_{i+1} . So in the top level of this skip list these elements remain. In gray we show the relationship between the skip list and a B+-tree that allows 4 elements per node.

38.4.2 Algorithmic Accessors

Almost all algorithmic accessors must first search for a tower holding the target element, or its insert position, within the skip list. While a linear time search in L_0 could be performed, the benefit of a skip list is that the additional structure provided by the index levels greatly reduces the search time. To find a target element in a skip list, the search begins at list $L_{\text{height}-1}$, by finding the towers *left* and *right* such that the target element is between them (according to the comparator). Then the search moves to the next level down, and repeats this process until L_0 is reached. Before presenting a more detailed description of the *findFirstOccurrence* and *findLastOccurrence* methods, we describe the relationship between the structure that supports search in a skip list with the structure that supports search in a binary search tree (Figure 38.4), and in a B+-tree (Figure 38.5).

First consider the skip list shown in Figure 38.4, in which exactly every other element in L_i is placed in L_{i+1} . The light gray lines in this figure illustrate the relationship between the structure of a binary search tree and the structure of a skip list. Consider the search process. In a binary search tree, first a comparison is made between the target element and the 4, the (only) element in L_2 , which corresponds to the root in a binary search tree. For any target element less than 4, the search continues in the subcollection that precedes the tower holding the 4. For any target element greater than 4, the search proceeds in the subcollection that follows the tower holding 4. This process is repeated within the subcollection at the next level. The main difference is that the skip list search introduces some extra checks to recognize that it has reached the tower that marks the right end of the remaining subcollection to search.

Next consider the skip list shown in Figure 38.5, in which exactly every fourth element in L_i is placed in L_{i+1} . The light gray boxes show the corresponding B+-tree. Observe that the skip list search method generalizes the B+-tree search method. The number of elements that must be examined at each level is the same as the number of elements in the corresponding B+-tree node. Again, an extra check is needed in the skip list to detect reaching the end of that sublist, as compared to a B+-tree where the number of elements in each node is known. This figure also illustrates the important difference between how elements are grouped in a skip list as compared to a B+-tree. In a B+-tree, each node contains a contiguous set of elements at a given level, whereas in a skip list each node contains a vertical segment corresponding to all occurrences of an element (at the leaf level list and the index level lists). An advantage of the B+-tree is that its organization provides very good properties in terms of caching, and also the size of each sublist that must be examined at a level is known (since it corresponds to the number of elements in the node). On the other hand, an advantage of the skip list is that by randomly selecting the height of each tower upon insertion, the expected structure, when viewed as B+-tree, is nearly balanced and no restructuring is needed.

We now describe the *findFirstOccurrence* method that takes *target*, the element to locate. It returns a reference to the first tower in the iteration order that holds an element equivalent to the target (if any). If there is no equivalent element in the collection, then this method is guaranteed to return the tower holding the successor of the target.


```

Tower<E> findFirstOccurrence(E target) {
    Tower<E> left = head;           //leftmost tower of range to search
    Tower<E> right = tail;          //rightmost tower of range to search
    int level = height - 1;         //start search at max level in use
    while (level ≥ 0) {
        Tower<E> next = left.next(level);           //tower after left (at level)
        if (right == next)                          //no towers between left and right (at level)
            level--;                                //so drop down a level
        else {
            if (comp.compare(target, next.element) > 0) //target > next tower's element
                left = next;                       //then update leftmost tower of range
            else {                                   //target < next tower's element
                right = next;                       //then rightmost tower of range moves
                level--;                             //and drop down a level
            }
        }
    }
    return right;
}

```

Correctness Highlights: We maintain the invariant that $left.element < target \leq right$. This invariant holds initially by REACHABLE since $left = head$ and $right = head$. We now argue that it is maintained when either $left$ or $right$ is updated. The update $left = next$ is only made when $target > next.element$. Thus, by SORTED, this invariant is maintained. Similarly, the update $right = next$ is made only when $target \leq next.element$. Again, by SORTED, this invariant is maintained.

At each level, as long as $target > next.element$, $left$ is shifted right. When $target \leq next.element$, $right$ is set to $next$. Thus, whenever $level$ is decremented, $left$ is the predecessor of $right$ at the current level. Combined with the invariant proven above, it follows that when $level$ is decremented to -1, if $target$ is in the collection, then $right$ references the leftmost tower holding $target$. Otherwise, $right$ holds the successor of $target$. ■

We now describe the *findLastOccurrence* method that takes $target$, the element to locate. It returns a reference to the last tower in the iteration order that holds an element equivalent to the target (if any). If there is no equivalent element in the collection, then this method is guaranteed to return the tower holding the predecessor of the target.

```

Tower<E> findLastOccurrence(E target) {
    Tower<E> left = head;           //leftmost tower of range to search
    Tower<E> right = tail;          //rightmost tower of range to search
    int level = height - 1;         //start search at max level in use
    while (level ≥ 0) {
        Tower<E> next = left.next(level);           //next tower at level
        if (right == next)                          //no towers between left and right at level
            level--;                                //so drop down a level
        else {
            if (comp.compare(target, next.element) ≥ 0) //target > next tower's element
                left = next;                       //then update leftmost tower of range
            else {                                   //target < next tower's element
                right = next;                       //then rightmost tower of range moves
                level--;                             //and drop down a level and continue
            }
        }
    }
    return right;
}

```

```

    }
  }
}
return left;           //target (if there) or otherwise predecessor
}

```

Correctness Highlights: The correctness is like that for *findFirstOccurrence* except it maintains the invariant that $left.element \leq target < right$. For this method, when *level* is decremented to -1, if *target* is in the collection, then *left* references the rightmost tower holding *target*. Otherwise, *left* holds the predecessor of *target*. ■

The method *contains* takes *target*, the element being tested for membership in the collection, and returns *true* if and only if an equivalent value exists in the collection.

```

public boolean contains(E target) {
    Tower<E> t = findFirstOccurrence(target);
    return (t != tail && comp.compare(t.element, target) == 0);
}

```

Correctness Highlights: By the correctness of *findFirstOccurrence*, if *target* is in the collection then *t* will be a reference to a tower holding an equivalent element. Otherwise, it will reference the successor of *target*. If *t* == *tail* then *target* is larger than all elements in the collection, and thus the correct answer is returned. ■

The method *get* takes *r*, the desired rank. It returns the r^{th} element in the sorted order, where $r = 0$ is the minimum. It throws an *IllegalArgumentException* when $r < 0$ or $r \geq n$.

```

public E get(int r) {
    if (r < 0 || r ≥ getSize())
        throw new IllegalArgumentException();
    Tower<E> ptr = head.next(0);
    for (int j=0; j < r; j++, ptr = ptr.next(0));
    return ptr.element;
}

```

Correctness Highlights: By SORTED and SUBSET, moving forward *r* times from *head.next(0)* leaves *ptr* at the rank *r* element. ■

The method *getEquivalentElement* takes *target*, the element for which an equivalent element is sought, and returns an equivalent element in the collection. It throws a *NoSuchElementException* when there is no equivalent element in the collection.

```

public E getEquivalentElement(E target) {
    Tower<E> t = findFirstOccurrence(target);
    if (t != tail && comp.compare(t.element, target) == 0)
        return t.element;
    else
        throw new NoSuchElementException();
}

```

Correctness Highlights: By the correctness of *findFirstOccurrence*, the target is in the collection exactly when the *tail* is not returned. ■

The *min* method returns a least element in the collection. It throws a *NoSuchElementException* when the collection is empty.

```
public E min() {
    return head.next(0).element;
}
```

Correctness Highlights: By SORTED and SUBSET, the first element in L_0 is the minimum element in the collection. ■

The *max* method returns a greatest element in the collection. It throws a *NoSuchElementException* when the collection is empty.

```
public E max() {
    return tail.prev(0).element;
}
```

Correctness Highlights: By SORTED and SUBSET, the last element in L_0 is the maximum element in the collection. ■

The *predecessor* method takes *target*, the element for which to find the predecessor. It returns the largest element in the ordered collection that is less than *target*. This method does not require that *target* be in the collection. It throws a *NoSuchElementException* when no element in the collection is smaller than *target*.

```
public E predecessor(E target) {
    Tower<E> t = findFirstOccurrence(target);
    t = t.prev(0);
    if (t == head)
        throw new NoSuchElementException();
    return t.element;
}
```

Correctness Highlights: By the correctness of *findFirstOccurrence*, if there is an element equivalent to the target in the collection, then *t* references a tower holding an equivalent element. Otherwise *t* references the successor of the target. In both cases, the tower that precedes *t* at level 0 holds the predecessor of the target. However, if *t* is head, there is no element in the collection less than *target*. ■

The *successor* method takes *target*, the element for which to find the successor. It returns the smallest element in the ordered collection that is greater than *target*. This method does not require that *target* be in the collection. It throws a *NoSuchElementException* when no element in the collection is greater than *target*.

```

public E successor(E target) {
    Tower<E> t = findLastOccurrence(target);
    t = t.next(0);
    if (t == tail)
        throw new NoSuchElementException();
    return t.element;
}

```

Correctness Highlights: By the correctness of *findLastOccurrence*, if there is an element equivalent to the target in the collection, then *t* references a tower holding an equivalent element. Otherwise *t* references the predecessor of the target. In both cases, the tower that follows *t* at level 0 holds the successor of the target. However, if *t* is *head*, there is no element in the collection greater than *target*. ■

38.4.3 Representation Mutators

When new towers are inserted for new elements, occasionally a new tower is taller than the head and tail. In that case, we need to grow the head and tail to accommodate more index levels. The internal *resizeHeadAndTail* method takes *minimumHeight*, the minimum acceptable height for *head* and *tail*. It resizes *head* and *tail* to be no more than twice that of *minimumHeight* while maintaining the same collection.

```

void resizeHeadAndTail(int minimumHeight) {
    int oldHeight = head.getTowerHeight();
    int newHeight = oldHeight * 2;
    while (newHeight < minimumHeight)
        newHeight *= 2;

    Tower<E>[] newHeadLinks = (Tower<E>[]) new Tower[newHeight*2];
    System.arraycopy(head.links, 0, newHeadLinks, 0, oldHeight);
    Arrays.fill(newHeadLinks, oldHeight, newHeight, tail);
    head.links = newHeadLinks;

    Tower<E>[] newTailLinks = (Tower<E>[]) new Tower[newHeight*2];
    System.arraycopy(tail.links, oldHeight, newTailLinks, newHeight, oldHeight);
    Arrays.fill(newTailLinks, newHeight + oldHeight, newHeight*2, head);
    tail.links = newTailLinks;

    head.setPrev(0, head);
}

```

//satisfy InUse

Correctness Highlights: As specified, this method modifies the height of the *head* and *tail* to be between *oldHeight* and $2 * \text{neededHeight}$. SUBSET and SORTED are maintained, since the non-empty lists are modified to be referenced by the next reference of the new head and the previous reference of the new tail. PREVIOUSREFERENCE is preserved by *setNext*. The line after the second for loop preserves INUSE. REDIRECTCHAIN is preserved since there is no change to which towers are reachable. INUSE must be preserved by the *insert* method. ■

38.4.4 Content Mutators

Methods to Perform Insertion

The internal method *biasedCoinFlip* method returns *true* with probability 1/4.

```
final boolean biasedCoinFlip() {
    return (randseq.nextBoolean() & randseq.nextBoolean());
}
```

The internal method *selectTowerHeight* returns the height for a new tower which is randomly selected as the number of biased coin flips made until a *true* is obtained, which occurs with probability 1/4. This method ensures that 1/4 of the elements in L_i are placed in L_{i+1} . So the probability that a tower has height h is $\left(\frac{1}{4}\right)^{h-1} \left(\frac{3}{4}\right)$. So

i	probability that height is i
1	$3/4 = 0.75$
2	$3/16 = 0.1875$
3	$3/64 = 0.046875$
4	$3/256 = 0.01171875$
5	$3/1024 = .0029296875$

```
int selectTowerHeight() {
    int num = 1;
    while (biasedCoinFlip() && num < MAX_HEIGHT)
        num++;
    return num;
}
```

The internal *insert* method takes *element*, the new element, and inserts *element* into the collection. It returns a reference to the newly created tower that holds the given element.

```
protected Tower<E> insert(E element) {
    int towerHeight = selectTowerHeight(); //pick random height
    int headHeight = head.getTowerHeight();
    if (towerHeight > height) {
        height = towerHeight; //maintain Height property
        if (towerHeight > headHeight) //must resize if new tower higher
            resizeHeadAndTail(towerHeight); //than current head and tail
    }
    Tower<E> newTower = new Tower<E>(element, towerHeight);
    Tower<E> left = head; //left boundary of search range (initially head)
    Tower<E> right = tail; //right boundary of search range (initially tail)
    int level = height - 1; //current level (start at max level in use)
    while (level >= 0) { //until done with bottom level
        Tower<E> next = left.next(level); //next tower at current level
        if (next != right && comp.compare(element, next.element) > 0)
            left = next; //left marker moves to right
        else { //element <= next elem.

```

```

        right = next;                                //right marker moves to left
        if (level < towerHeight) { //splice in newTower between left and right at this level
            left.setNext(level, newTower);
            newTower.setNext(level, right);
        }
        level--;                                     //move down to next level
    }
}
size++;                                           //increment size to maintain Size property
return newTower;
}

```

Correctness Highlights: Inserting an element in the collection can only increase the height. Thus, the conditional *if (towerHeight > height)* preserves INUSE. By the correctness of *resize-HeadAndTail*, when *newTower* is allocated, all properties hold.

Since the height of the new tower is at least 1, it will be inserted into L_0 , so REACHABLE is preserved. Observe that once *level < towerHeight*, the new element is inserted into L_{level} . Since *towerHeight* does not change once it is assigned, and *level* is decremented during each execution of the main loop, if the element is inserted in L_i , it will be inserted in L_j for all $0 \leq j < i$, so SUBSET is maintained.

The *insert* method maintains the loop invariant that *left.element* $\leq$ *element* $\leq$ *right.element*. This invariant holds initially, since by construction *head.element* is smaller than all possible elements, and *tail.element* is greater than all elements. It is easy verified that the updates made during the loop preserve this invariant. The new element is only spliced into L_i when *left.next[i] = right*. Thus, SORTED is maintained.

HEIGHT is maintained by the tower *setNext* method, and SIZE is preserved by incrementing *size* by 1. INUSE and REDIRECTCHAIN are not changed by this method, so they continue to hold. ■

The *add* method takes *element*, the new element, and inserts *element* into the collection.

```

public void add(E element) {
    insert(element);
}

```

Correctness Highlights: The correctness follows from that of *insert*. ■

The *addTracked* method takes *element*, the new element and inserts *element* into the collection. It returns a tracker that tracks the new element.

```

public Locator<E> addTracked(E element) {
    return (Locator<E>) new Tracker(insert(element));
}

```

Correctness Highlights: The correctness follows from that of *insert*. ■

The efficiency of the *addAll* method can be improved when *c* is an ordered collection. The basic idea is very similar to the merge process used in merge sort (Section 11.4.2), except that the new tower may need to be inserted into multiple levels. Since the expected sum of the sizes of all lists is $O(n)$, the overall cost is linear.

The internal *merge* method takes *c*, the collection for which the elements are to be merged into this skip list. It requires that *c* is an ordered collection. The array *succ* of tower references is maintained so that *succ[i]* references the successor of the current position in L_i . Since a new random height is selected for each new tower, it is possible that the head and tail, and also *succ* need to be resized. After any needed resizing is performed, the new tower is created to hold the next element from *c*. Using L_0 , the location for the new tower is located. Also, whenever moving forward, the next references at all levels of the current tower are copied into *succ*. By INORDER it follows that when the location for the new tower is located in L_0 , *succ[i]* references the tower before which the new tower should be placed at each level. Then the new tower is spliced into L_i immediately before *succ[i]*, at all levels that exist within it. If there are equivalent elements, the new tower will be placed just before them. If desired, *succ* could move forward at an equivalent element resulting in the new element being added after any equivalent elements.

```

void merge(Collection<? extends E> c) {
    int headHeight = head.getTowerHeight();
    Tower<E>[] succ = new Tower[headHeight];
    System.arraycopy(head.links, 0, succ, 0, headHeight);
    for (E element: c) {
        int towerHeight = selectTowerHeight();           //select height for new tower
        if (towerHeight > height) {
            height = towerHeight;                       //maintain Height property
            if (towerHeight > head.getTowerHeight()) {  //must resize if new tower higher
                resizeHeadAndTail(towerHeight);
                succ = resizePtr(succ);
            }
        }
        Tower<E> newTower = new Tower<E>(element, towerHeight); //create new tower
        while (succ[0] != tail && comp.compare(succ[0].element, element) < 0)
            System.arraycopy(succ[0].links, 0, succ, 0, succ[0].getTowerHeight());
        for (int i = 0; i < towerHeight; i++) {           //splice in new tower at each level
            succ[i].prev(i).setNext(i, newTower);        //insert new tower in  $L_i$  just before succ[i]
            newTower.setNext(i, succ[i]);
        }
    }
}

```

Correctness Highlights: By the correctness of *resizeHeadAndTail* and *resizePtr*, the head, tail, and *succ* arrays will have an entry for L_i for $0 \leq i < \text{towerHeight}$. Also, when resizing occurs, the instance variable *height* is updated to preserve HEIGHT.

By the requirements of this method, the iteration order for *c* is in sorted order. Let *t* be the tower referenced by *succ[0]* just prior to the update made in the body of the while loop. By SORTED, it follows that *succ[i]* is the successor of *t.element* in the collection. Observe, that this holds for levels that do not contain *t.element*. Since *succ[0]* advances until the element that follows it is greater than or equal to the element being added, it follows that inserting the new element into L_i immediately before *succ[i]*, preserves SORTED. Finally, since the elements in *c* are considered in sorted order, it is guaranteed, that the proper insert position for the next element will be at or after the current position. ■

The *resizePtr* method takes *succ*, an array of tower references. It resizes *succ* so that it has the same capacity as the head. It requires that no new elements have been added since the head and tail were resized.

```

Tower<E>[] resizePtr(Tower<E>[] succ) {
    int headHeight = head.getTowerHeight();
    Tower<E>[] newPtr = new Tower[headHeight];
    System.arraycopy(succ, 0, newPtr, 0, succ.length);
    Arrays.fill(newPtr, succ.length, headHeight, tail);
    return newPtr;
}

```

Correctness Highlights: By the requirement of this method, at all new levels *tail* immediately follows *head*. Thus for any possible element, *tail* is the successor, so the property required of *succ* is preserved. For all existing levels, the old value is copied into the new array. ■

Recall the *addAll* method takes *c*, the collection to be added. This method iterates through *c* and adds each element in *c* to the collection. If *c* is an ordered collection with the same comparator then the linear time *merge* method can be used. Otherwise, successive insertions are used (by the superclass *addAll* method).

```

public void addAll(Collection<? extends E> c) {
    if (c instanceof OrderedCollection && c.getComparator().equals(getComparator()))
        merge(c);
    else
        super.addAll(c);
}

```

Correctness Highlights: Follows from the correctness of *merge* that requires that *c* be an ordered collection, and the correctness of the superclass *addAll* that does not place any requirements on *c*. ■

Methods to Perform Deletion

The internal *remove* method takes *t*, a reference to a tower in the collection, and removes it.

```

void remove(Tower<E> t) {
    int towerHeight = t.getTowerHeight();
    for (int i = 0; i < towerHeight; i++)
        t.prev(i).setNext(i, t.next(i));           //remove from all levels it's in
    if (height == towerHeight){                     //see if skip list height decreases
        while (height > 1 && head.next(height) == tail) //preserves Height property
            height--;
    }
    t.delete();                                     //preserves InUse property
    size--;                                         //preserves Size property
}

```

Correctness Highlights: REACHABLE is preserved by the *for* loop, which removes tower *t* from all lists that contain it. SORTED is preserved since no elements are added or reordered. The *setNext* method preserves PREVIOUSREFERENCE. If the tower removed was at the highest level

in use, then the *while* loop preserves HEIGHT by setting *height* to the maximum level in use. The *delete* method preserves INUSE. Finally, REDIRECTCHAIN is preserved, since by SORTED, *t.next[0]* references the successor of *t*. Finally, SIZE is preserved by decrementing *size*. ■

The public *remove* method takes *element*, the element to remove. It removes the first occurrence of the target from the collection, if an equivalent element exists in the collection. It returns *true* if an element was removed, and *false* otherwise.

```
public boolean remove(E element) {
    Tower<E> t = findFirstOccurrence(element);
    if (t == tail || comp.compare(t.element, element) != 0)
        return false;
    remove(t);
    return true;
}
```

Correctness Highlights: By the correctness of *FindFirstOccurrence* if there is an element in the collection equivalent to *element*, then *t* references the tower with the first occurrence of that element. Otherwise, *t* references the tower holding the successor. Thus, if *t* is at *tail* then there is no equivalent element in the collection. The rest of the correctness follows from that of *remove(t)*. ■

Recall that the *retainAll* method takes *c*, a collection, and updates the current collection to contain only elements that are also in *c*. When *c* is an ordered collection with the same comparator, this method takes advantage of that fact for improved efficiency.

```
public void retainAll(Collection<E> c) {
    if (isEmpty()) //special case for efficiency
        return;
    if (c instanceof OrderedCollection && c.getComparator().equals(getComparator())) {
        Tower<E> ptr = head.next(0);
        for (E element: c) { //c is sorted
            while (ptr != tail && comp.compare(element, ptr.element) > 0) {
                remove(ptr);
                ptr = ptr.next(0);
            }
            while (ptr != tail && comp.compare(element, ptr.element) == 0)
                ptr = ptr.next(0);
        }
    }
    else
        super.retainAll(c);
}
```

Correctness Highlights: The collection does not change when it is empty. We now consider when the collection is not empty. First we consider when *c* is an ordered collection with the same comparator. Since both *c* and this collection are sorted, as long as *ptr.element* is smaller than the current element being considered from *c*, it is known that there is no equivalent element in *c*, and so it can be removed. As long as *ptr.element* is equivalent to the current element in *c*, *ptr* advances, properly leaving that element in this collection. Finally, when an element in this

collection is reached that is larger than the current element from *c*, the for loop advances to the next element from *c*. Thus upon completion of the for loop, the elements that remain in this collection are exactly those that were also in *c*.

When *c* is not an ordered collection or uses a different comparator, the correctness follows from that of the abstract collection *retainAll* method. ■

38.4.5 Locator Initializers

The abstract *iterator* method creates a new tracker at FORE.

```
public Tracker iterator() {
    return new Tracker(head);
}
```

Correctness Highlights: The head sentinel immediately precedes the first element in the iteration order. ■

The abstract *iteratorAtEnd* method creates a new tracker at AFT.

```
public Tracker iteratorAtEnd() {
    return new Tracker(tail);
}
```

Correctness Highlights: The tail sentinel immediately follows the last element in the iteration order. ■

The abstract *Locator* method takes *element*, an element to locate. It returns a tracker to the specified element. It throws a *NoSuchElementException* when there is no equivalent element in the ordered collection.

```
public Tracker getLocator(E element) {
    Tower<E> t = findFirstOccurrence(element);
    if (t == tail || comp.compare(element, t.element) != 0)
        throw new NoSuchElementException();
    return new Tracker(t);
}
```

Correctness Highlights: Follows from the correctness of *findFirstOccurrence* (along with the observation that if *tail* is returned then *element* is not in the collection) and the Tracker constructor. ■

38.5 Tracker Inner Class

AbstractCollection.AbstractLocator implements *Locator*

↳ **Tracker** implements *Locator*

Each skip list has an instance variable *loc* that refers to the tower holding the tracked element.

```
Tower<E> loc; //reference to the tracked tower
```

The constructor takes a single argument, *loc*, a reference to the tower to track.

```
Tracker(Tower<E> loc) {
    this.loc = loc;
}
```

Recall that *inCollection* returns *true* if and only if the tracked element is currently in the collection.

```
public boolean inCollection() {
    return !loc.isDeleted();
}
```

Correctness Highlights: Follows from INUSE, and the correctness of the Tower *isDeleted* method. ■

The *get* method returns the tracked element. It throws a *NoSuchElementException* when tracker is not at an element in the collection.

```
public E get() {
    if (!inCollection())
        throw new NoSuchElementException();
    return loc.element;
}
```

Next we consider the methods for moving a tracker forward (*advance*) or backward (*retreat*) in the iteration order. These methods must handle the situation in which the tracked tower is no longer in the collection. In such cases, we know from REDIRECTCHAIN that the first tower in the collection reached when following the chain of level 0 next references is the successor (or *tail* if the tracker has no successor).

The internal *skipRemovedElements* method takes *t*, a reference to a tower, and returns a reference to the first tower in the collection reached starting at *t* and following the L_0 next references. If there is no such element, then *tail* is returned. For the sake of efficiency, path compression is performed. Namely, if a sequence of *next* pointers in L_0 is followed for a sequence of unreachable towers, then once a reachable tower is found, all of the L_0 *next* pointers followed are updated to refer to the tower that was reached.

```
private Tower<E> skipRemovedElements(Tower<E> t) {
    if (!t.isDeleted())
        return t;
    if (t.next(0).isDeleted())
        t.links[0] = skipRemovedElements(t.next(0));
    return t.next(0);
}
```

Correctness Highlights: If t references an element in the collection, or *head* or *tail*, it is the correct return value. Otherwise, once an element in the collection is reached by following the L_0 next references, by REDIRECTCHAIN, it holds the next tower in the iteration order. Setting $t.links[0]$ to the returned value performs the path compression. Termination is guaranteed by REDIRECTCHAIN. ■

The *advance* method moves the tracker to the next element in the iteration order (or *tail* if the last element is currently tracked). It returns *true* if and only if after the update, the tracker is already at an element of the collection. It throws an *AtBoundaryException* when the tracker is at *tail* since there is no place to advance.

```
public boolean advance() {
    if (loc == tail)
        throw new AtBoundaryException();
    if (loc.isDeleted())
        loc = skipRemovedElements(loc);
    else
        loc = loc.next(0);
    return loc != tail;
}
```

Correctness Highlights: If the tracker is currently at *tail*, an *AtBoundaryException* is properly thrown. If the tracked element is no longer in the collection, by the correctness of *skipRemovedElements* it is moved to the successor. If the element is in the collection, then by SORTED (and REACHABLE and SUBSET that imply all elements in the collection are in L_0), the next element in the iteration order is the next element in L_0 . Finally, the tracker is at an element in the collection as long as it has not reached *tail*. ■

The *retreat* method moves the tracker to the previous element in the iteration order (or *head* if the first element is currently tracked). It returns *true* if and only if after the update, the tracker is at an element of the collection. It throws an *AtBoundaryException* when the tracker is at *head* since then there is no place to retreat. Because *skipRemovedElements* returns the next element in the iteration order for an element no longer in the collection, the *advance* method is slightly simpler. In implementing *retreat*, we must address the situation in which new elements have been added to the collection that are between the tracker location and the element returned by *skipRemovedElements*. For example, let “ e ,” the tracked element in the collection, have predecessor “ b ” and successor “ s .” Suppose that e is removed, leaving the tracker logically between b and s . Suppose that elements “ p ,” “ q ,” and “ r ” are then added to the collection. The proper return value for *retreat* is “ b .” However, the return value from *pred(skipRemovedElements(node))* would be “ r ” since it is the current predecessor of “ s .” However, the added elements fall in the iteration order after to the tracker and so the correct return value should be “ b .” The *while* loop addresses this situation.

```
public boolean retreat() {
    if (loc == head)
        throw new AtBoundaryException();
    E oldData = loc.element;
    boolean wasDeleted = loc.isDeleted();
    if (wasDeleted)
        loc = skipRemovedElements(loc);
    loc = loc.prev(0);
    if (wasDeleted && loc != head)
```

```

        while (comp.compare(loc.element, oldData) > 0)
            loc = loc.prev(0);
    return loc != head;
}

```

Correctness Highlights: If *node* is at *head* then an *AtBoundaryException* is properly thrown. If the tracked element had been deleted, then by the correctness of *skipRemovedElements* the tracker is moved to be logically just be the successor of the element at the time when it was removed. By REACHABLE, and SUBSET, all elements are in L_0 . Furthermore, by SORTED, *prev(0)* references to the previous tower in the iteration order. When *wasDeleted* is false, moving back one tower in the iteration order is guaranteed to correctly position the tracker. Now consider when the tower had been deleted. In this case, *loc* is at the predecessor of its current location. However, as discussed above, it could be at an element that was added which is larger than *oldData* and is therefore not the correct return value. The *while* loop continues to move back one element in the iteration order until reaching a tower in the iteration order with a value less than or equal to that of *oldData*. This tower is the correct node to be tracked.

Finally, *loc* is at an element in the collection as long as it is not at *head* after retreating. Thus, the correct value is returned. ■

The *hasNext* method returns *true* if there is some element after the current tracker position.

```

public boolean hasNext() {
    if (loc.isDeleted())
        loc = skipRemovedElements(loc);
    return loc.next(0) != tail;
}

```

Correctness Highlights: Like that for *advance*, except that some cases are not needed since the value of *loc* need not be updated. ■

As discussed in Section 5.8, the *remove* method removes the tracked element and updates the tracker to be at the element in the iteration order that preceded the one removed. It throws a *NoSuchElementException* when the tracker is at FORE or AFT.

```

public void remove() {
    SkipList.this.remove(loc);
}

```

Correctness Highlights: Follows from the skip list internal *remove* method that takes a tower reference as its parameter. ■

38.6 Performance Analysis

The expected asymptotic time complexity of each public method of the *Skiplist* class is shown in Table 38.6, and the expected asymptotic time complexity for each public method of the *SkipListTracker* class is given in Table 38.7.

method	time complexity
constructors	$O(1)$
<i>ensureCapacity(x)</i>	$O(1)$, worst case
<i>iterator()</i>	$O(1)$, worst case
<i>iteratorAtEnd()</i>	$O(1)$, worst case
<i>max()</i>	$O(1)$, worst case
<i>min()</i>	$O(1)$, worst case
<i>trimToSize()</i>	$O(1)$, worst case
<i>add(o)</i> , <i>addTracked(o)</i>	$O(\log n)$, expected
<i>contains(o)</i>	$O(\log n)$, expected
<i>getEquivalentElement(o)</i>	$O(\log n)$, expected
<i>getLocator(o)</i>	$O(\log n)$, expected
<i>predecessor(o)</i>	$O(\log n)$, expected
<i>remove(o)</i>	$O(\log n)$, expected
<i>successor(o)</i>	$O(\log n)$, expected
<i>accept(v)</i>	$O(n)$, worst case
<i>clear()</i>	$O(n)$, worst case
<i>toArray()</i>	$O(n)$, worst case
<i>toString()</i>	$O(n)$, worst case
<i>retainAll(c)</i>	$O(n c)$
<i>addAll(c)</i>	$O(c \log(n + c)$, expected

Table 38.6 Summary of the asymptotic time complexities for the public methods when using the skip list data structure to implement the OrderedCollection ADT.

locator method	time complexity
constructor	$O(1)$, worst case
<i>get()</i>	$O(1)$, worst case
<i>advance()</i>	$O(1)$, worst case
<i>retreat()</i>	$O(1)$, worst case
<i>next()</i>	$O(1)$, worst case
<i>hasNext()</i>	$O(1)$, worst case
<i>remove()</i>	$O(1)$, expected

Table 38.7 Summary of the time complexities for the public locator methods of the skip list locator class. If elements have been removed, then these methods have constant amortized cost because of the computation performed by *skipRemovedElements*, and also the need for *retreat* to possibly move past newly inserted elements.

We first derive the expected time to search for an element, or its insert position, in a skip list. The rest of the analysis follows fairly directly from this. We closely follow the original analysis of Pugh [127]. It is important to note that the expectation is just with respect to the randomization performed within the skip list insertion method. No assumptions are made about the elements inserted, or the order of insertion.

We prove inductively that at level i , we expect $n \cdot p^i$ elements. At the bottom level ($i = 0$), there are n elements. For the inductive step, observe that for each element in L_i , with probability p it is placed in L_{i+1} . Thus, if there are x elements in L_i , $E[|L_{i+1}|] = x \cdot p$. Combined with the inductive hypothesis yields that $E[|L_{i+1}|] = n \cdot p \cdot p^{i-1} = n \cdot p^i$. We define $L(n)$ as the level i for which $E[|L_i|] = 1/p$. Solving for $L(n)$ in $n \cdot p^{L(n)} = 1/p$, yields

$$L(n) = \log_{\frac{1}{p}} n - 1.$$

We now compute the expected number of steps to reach a target element in L_0 . Until reaching level 0, at each step with probability $1 - p$, the search moves to the right at the current level, and with probability p , the search moves down a level. Let $C(k)$ be the expected number of steps in the search path starting at level k . From the above discussion, it follows that:

$$C(k) = \begin{cases} 0, & \text{if } k = 0 \\ (1-p)(1 + C(k)) + p(1 + C(k-1)) & \text{otherwise.} \end{cases}$$

Solving for $C(k)$ yields that $C(k) = k/p$. So the expected search cost starting at level $L(n)$ is $L(n)/p = (\log_{1/p} n - 1)/p$. The expected number of additional levels [127] in the skip list is $1 + \frac{1}{1-p}$, yielding that

$$E[\text{height}] = L(n) + 1 + \frac{1}{1-p} = \log_{\frac{1}{p}} n + \frac{1}{1-p},$$

where *height* is the maximum level of any tower.

Thus the expected number of steps in the search path starting from $L_{\text{height}-1}$ is

$$\frac{L(n)}{p} + \frac{1}{1-p} + 1 = O(\log n).$$

Papadakis, Munro, and Poblete [121] proved that the expected number of steps to find the i^{th} smallest element in the ordered collection is

$$\frac{1}{p} \log_{\frac{1}{p}} i + \log_{\frac{1}{p}} \frac{n}{i} + O(1)$$

where they gave a precise expression for the $O(1)$ term. It has also been shown that the search cost is minimized when $p = 1/e$ where e is Euler's constant.

For any given tower, let h be its height. The expected value for h is the number of biased coin flips until a head is obtained where a head occurs with probability $1 - p$ and a tail occurs with probability p . Using the standard expected value analysis for a binomial distribution (e.g., [52]),

$$E[h] = \sum_{i=0}^{\infty} i p^{i-1} (1-p) = \frac{1}{1-p}.$$

Thus the total number of references in the *links* arrays for all towers in the collection is $2n/(1-p)$. Finally the head and tail sentinels have height at most $2E[\text{height}]$. Thus the expected number of references, which dominates the space usage, is at most

$$\frac{2n}{1-p} + 4 \left(L(n) + 1 + \frac{1}{1-p} \right) = \frac{2n}{1-p} + 4 \log_{\frac{1}{p}} n + \frac{4}{1-p}.$$

	As a function of p	$p = 1/2$	$p = 1/e$	$p = 1/4$
space usage	$2n/(1-p) + 4 \log_{1/p} n$	$4n + 4 \log_2 n$	$\approx 3.2n + 2.8 \log_2 n$	$\approx 2.67n + 2 \log_2 n$
search cost	$(\log_{1/p} n)/p$	$2 \log_2 n$	$\approx 1.88 \log_2 n$	$2 \log_2 n$

Table 38.8 A comparison of the space usage and search cost as p varies. The expected space usage, measured in the number of tower references, excludes an additive $4/(1-p)$ term. The expected search cost, measured in the expected number of tower references followed, excludes an additive term of $\frac{2-p}{1-p} - \frac{1}{p}$.

In our implementation we fix $p = 1/4$. However, in general, different values for p could be selected. Table 38.8 compares the space usage and expected search cost for several values for p . If the application does not need efficient support for *retreat* then the implementation can be modified to not use the *prev* links, which would cut the space usage almost in half. Recall that for a red-black tree, in the worst case $2 \log_2 n$ child references are followed in a search, and if the elements are inserted in a random order then the expected search cost (measured in child references to follow) is close to $\log_2 n$. Since each red-black tree node has a left child, right child, and parent references, there are $3n$ references stored in addition to the boolean maintained to hold the color. If the nodes were threaded to support constant time iteration (as supported by a skip list), then the space usage would be $5n$ references.

Clearly the constructors take worst-case constant time. Since L_0 is a sorted list with all elements in the collection, *min* and *max* have worst-case constant cost. Likewise, the locator constructor runs in worst-case constant time, so *iterator* and *iteratorAtEnd* take constant time. Also, since the skip list is an elastic implementation, *ensureCapacity* and *trimToSize* need not perform any computation, so they take constant time. The *accept*, *clear*, *toArray*, and *toString* methods take linear time since L_0 is a sorted list.

The expected cost of *findFirstOccurrence* is logarithmic since you can view it as searching for an element that is just smaller than the target. Likewise, the expected cost of *findLastOccurrence* is logarithmic. Thus *contains* and *getLocator* have expected logarithmic cost. Also, since L_0 is a sorted list, it follows that *predecessor* and *successor* have expected logarithmic cost.

The internal *insert* method used by both *add* and *addTracked* is a variation of the standard search method that inserts each tower in its proper location in any list that includes it before moving down a level. Thus *add* and *addTracked* have expected logarithmic time. Finally, once an element is located, it can be removed in expected constant time since we have argued that the expected height of a tower is $1/(1-p)$. Thus *remove* also has expected logarithmic cost.

When *addAll* is called with an ordered collection c , it uses a single pass through L_0 , to locate where each element from c is to be inserted. Since the expected height of a tower is $1/(1-p)$, the expected cost for each insertion is constant. Thus the expected time complexity is $O(n + |c|)$. When c is not an ordered collection, *addAll* method performs $|c|$ insertions into a skip list that has a final size of $O(n + |c|)$. Thus, the expected time complexity is $O(|c| \log(n + |c|))$.

When *retainAll* is called with a parameter that is an ordered collection, it simultaneously iterates through c and n , just retaining the elements that are in c . Since the cost to remove an element from the skiplist (once located) is expected constant, the expected cost for retain all is $O(n + |c|)$. When *retainAll* is called with a collection that is not an ordered collection, for each element in this collection it must search for that element in c , which takes $O(|c|)$ time, and then possibly remove that element from this collection (through a locator), which takes expected constant cost. Thus, the overall expected time complexity is $O(n \cdot |c|)$.

We now analyze the time complexity of the locator methods. The methods that just access an element take $O(1)$ time. The *skipRemovedElements* method takes time proportional to the length

redirect chain. However, every element in the redirect chain is the result of *remove* being called (either directly or through a tracker). By charging each *remove* method call constant cost, the optimization performed by *skipRemovedElements* guarantees that it has constant amortized cost. All methods to move forward or backward in the iteration order take worst-case constant time (excluding the cost of *skipRemovedElements*) since L_0 is a sorted list. Finally, since the expected height of a tower is constant, *remove* has expected constant cost.

38.7 Quick Method Reference

SkipList Public Methods

- p. 599 *SkipList*()
- p. 599 *SkipList*(Comparator<? super E> comp)
- p. 599 *SkipList*(int size)
- p. 598 *SkipList*(int size, Comparator<? super E> comp)
- p. 98 void **accept**(Visitor<? super E> v)
- p. 606 void **add**(E element)
- p. 100 void **addAll**(Collection<? extends E> c)
- p. 606 Locator<E> **addTracked**(E element)
- p. 100 void **clear**()
- p. 97 boolean **contains**(E value)
- p. 99 void **ensureCapacity**(int capacity)
- p. 602 E **get**(int r)
- p. 96 int **getCapacity**()
- p. 96 Comparator<? super E> **getComparator**()
- p. 98 E **getEquivalentElement**(E target)
- p. 610 Tracker **getLocator**(E element)
- p. 96 int **getSize**()
- p. 96 boolean **isEmpty**()
- p. 610 Tracker **iterator**()
- p. 610 Tracker **iteratorAtEnd**()
- p. 603 E **max**()
- p. 603 E **min**()
- p. 603 E **predecessor**(E target)
- p. 609 boolean **remove**(E element)
- p. 100 void **retainAll**(Collection<E> c)
- p. 603 E **successor**(E target)
- p. 97 Object[] **toArray**()
- p. 97 E[] **toArray**(E[] array)
- p. 98 String **toString**()
- p. 99 void **trimToSize**()

SkipList Internal Methods

- p. 605 boolean **biasedCoinFlip**()
- p. 97 int **compare**(E e1, E e2)
- p. 97 boolean **equivalent**(E e1, E e2)
- p. 600 Tower<E> **findFirstOccurrence**(E target)
- p. 601 Tower<E> **findLastOccurrence**(E target)
- p. 605 Tower<E> **insert**(E element)
- p. 607 void **merge**(Collection<? extends E> c)
- p. 608 void **remove**(Tower<E> t)
- p. 604 void **resizeHeadAndTail**(int minimumHeight)

- p. 607** `Tower<E>[] resizePtr(Tower<E>[] succ)`
- p. 605** `int selectTowerHeight()`
- p. 99** `void traverseForVisitor(Visitor<? super E> v)`
- p. 98** `void writeElements(StringBuilder s)`

SkipList.Tower Internal Methods

- p. 597** `Tower(T element, int towerHeight)`
- p. 598** `void delete()`
- p. 597** `int getTowerHeight()`
- p. 598** `boolean isDeleted()`
- p. 597** `Tower<T> next(int level)`
- p. 597** `Tower<T> prev(int level)`
- p. 598** `void setNext(int level, Tower<T> t)`
- p. 597** `void setPrev(int level, Tower<T> t)`

SkipList.Tracker Public Methods

- p. 612** `boolean advance()`
- p. 611** `E get()`
- p. 613** `boolean hasNext()`
- p. 101** `void ignoreConcurrentModifications(boolean ignore)`
- p. 101** `void ignorePriorConcurrentModifications()`
- p. 611** `boolean inCollection()`
- p. 101** `E next()`
- p. 613** `void remove()`
- p. 612** `boolean retreat()`

SkipList.Tracker Internal Methods

- p. 611** `Tracker(Tower<E> loc)`
- p. 101** `void checkValidity()`
- p. 611** `Tower<E> skipRemovedElements(Tower<E> t)`
- p. 101** `void updateVersion()`

Chapter 39

Digitized Ordered Collection ADT

package collection.ordered.digitized

Collection<E>

└ *OrderedCollection*<E>

└ *DigitizedOrderedCollection*<E>

A digitized ordered collection is an untagged algorithmically positioned collection whose elements can each be viewed as a sequence of digits (e.g., bit string, character string). The collection cannot contain elements e_1 and e_2 where e_1 is a prefix of e_2 . By definition, e is a prefix of itself, so the collection may not contain duplicates. The *DigitizedOrderedCollection* interface extends the *OrderedCollection* interface by adding methods to find all extensions of a given prefix, and also to find all elements in the collection that have the longest prefix match with a given element.

While a digitized ordered collection extends an ordered collection, it can also be a good choice for applications that simply require searching for an equivalent element. For such applications, no ordering among the elements is used. There are two important advantages of a *DigitizedOrderedCollection* over a *Set* or *Mapping* ADT implementation based on hashing. The first is the ability to efficiently find all extensions of a given prefix or all elements sharing a longest common prefix. The second advantage is that often an unsuccessful search requires looking at only a small number of digits, whereas a hashing-based data structure uses all of the digits in computing the hash function and in making the comparison at each probe. We briefly describe an application that benefits from each of these advantages.

39.1 Case Study: Packet Routing

Maintaining an **Internet Protocol (IP) routing lookup table** requires the ability to efficiently find the longest prefix match for a desired destination address. An **IP address** is a 32 bit number allocated in hierarchical blocks. The first block contains the **destination address**. The destination itself is also divided into two portions. The first portion identifies the network on which the host resides, and the next portion identifies the particular host on the network. Such a design removes the need for a global address space to be stored in each router. The length of the prefix that identifies the network varies from packet to packet (e.g., for a router in the U.S., a packet destined for Europe would typically need to match a shorter prefix than one destined for elsewhere in the U.S., because all traffic to Europe would generally take the same path out of that router). Also observe that since all IP addresses have the same length, no IP address can be a prefix of another. Network routers maintain an IP routing lookup table which maps each IP address to an output port. When provided with a packet's destination IP address, the router must use the IP routing lookup table to efficiently determine on which output port to forward the packet. Any port whose associated address shares a common prefix with the packet's destination IP address will move the packet closer to its destination. However, there may be many ports with a shared common prefix and of those the one with the longest common prefix will move the packet as close as possible to the desired router. So the IP

router must efficiently find the IP address in the routing table that shares the longest common prefix with the packet's destination IP address. A similar approach is used within firewalls to check if the sender address has a common prefix with a collection of approved (or forbidden) IP addresses.

39.2 Case Study: Inverted Index for Text Retrieval

Another example application for a digitized ordered collection is an **inverted index** that associates, with every word in a dictionary, a list of documents (or web pages) containing that word. (The book by Salton and Buckley [129] provides a good overview of the information retrieval field which has developed and made use of inverted indexes for information retrieval.) So a mapping from words to a list of document IDs must be maintained. The number of words in the mapping could be extremely large, yet each individual word is generally fairly short. To ensure that no word is a prefix of another, a special “end of string” character (shown as #) is added to the end of each word.

An advantage of a digitized ordered collection is that the time to insert, remove, or search is linear in the number of characters in the word being considered. Furthermore, for such an application there are likely to be many unsuccessful searches (since query words may not appear in the dictionary). The cost of an unsuccessful search depends linearly on the length of the shortest prefix of the query word needed to distinguish it from all words in the dictionary. In contrast, computing a hash function requires computing a function that depends on each character in the query word.

Like the digitizer used for radix sort (Section 10.1), the constructor for each digitized ordered collection takes as a parameter a **digitizer** that is responsible for dividing the elements into digits. In particular, a digitizer provides a method *getDigit(x,p)* that returns the value of the p^{th} digit of element x . Our implementations can be applied to any type of element for which a digitizer can be provided. We also define the *PrefixFreeDigitizer* interface that extends the *Digitizer* interface to indicate enforcement of the property that no element in the collection is a prefix of another.

The iteration order for a digitized ordered collection is defined implicitly by the digitizer that is provided to its constructor. For digitizer d and distinct elements a and b , let p be the leftmost position (lowest index) at which $d.\text{getDigit}(a,p) \neq d.\text{getDigit}(b,p)$. We say that $a \prec b$ with respect to d if and only if $d.\text{getDigit}(a,p) < d.\text{getDigit}(b,p)$.

We provide, for illustration, an implementation of a *PrefixFreeDigitizer* over an alphabet of letters $a, \dots, z$. This digitizer satisfies the property that for unique elements e_1 and e_2 , $e_1 \prec e_2$ if and only if e_1 is lexicographically before e_2 . Observe that $a\#$ is lexicographically before $ate\#$, so the end of string character should precede all the digits in the alphabet. The example implementation maps the end of string character to the integer 0, the letter “a” to 1, the letter “b” to 2, and so on. The *base* is the number of letters in the alphabet, including the end of string character. If the alphabet consists of the end of string character and the letters $a, \dots, z$, then the base will be 27.

This string digitizer requires that all strings provided as arguments contain only the 26 lower case letters.

```
public class StringDigitizer implements PrefixFreeDigitizer<String> {
    int base;                                //base including the end of string character
```

The user provides the original alphabet size to the constructor. The resulting base is one larger because of the end of string character.

```
public StringDigitizer(int alphabetSize){base = alphabetSize + 1;}
```

The *getBase* method returns the base of the alphabet that includes the end of string character.

```
public int getBase() {return base;}
```

The *isPrefixFree* method returns *true* since this is a prefix free digitizer.

```
public boolean isPrefixFree() {return true;}}
```

The *numDigits* method takes *x*, the given string. It returns the number of digits in the string including the end of string character.

```
public int numDigits(String x) {return x.length() + 1;}
```

The *getDigit* method takes *x*, the given string, and *place*, the place of the desired digit, where the leftmost digit is place 0. It returns the integer value mapped to by the digit in the given place. This method implicitly pads all strings to the right with the end of string character by returning a 0 if the given place is past the end of the string *x*. For example, for the string *cat#*, when *place* = 0 the return value is 3 since “c” is the third letter in the alphabet. For *place* = 1 the return value is 1, and for *place* = 2 the return value is 20. For any value of *place* > 2, the return value is 0. So one can view this convention as padding all strings to the right with #. However, only the leftmost # is ever examined by any methods we introduce.

```
public int getDigit(String x, int place) {
    if (place ≥ x.length())
        return 0;
    else
        return x.toLowerCase().charAt(place) - 'a' + 1;
}
```

39.3 Digitized Ordered Collection Interface

The following methods are added to those inherited from the *OrderedCollection* interface (Chapter 29).

DigitizedOrderedCollection(): Creates a new empty digitized ordered collection using the default digitizer.

DigitizedOrderedCollection(Digitizer d): Creates a new empty string collection with the given digitizer.

void completions(E prefix, Collection<? super E> c): Appends all elements in this collection that have the given prefix to the given collection *c*. We consider an element to be a prefix of itself.

void longestCommonPrefix(E element, Collection<? super E> c): Appends to the provided collection *c* all elements in this collection that have a longest common prefix with *element*.

Critical Mutators: none

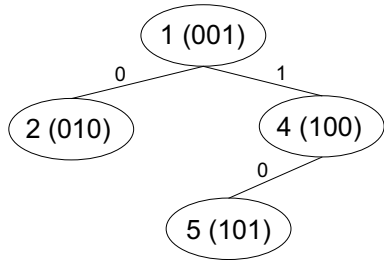


Figure 39.1
An example of a digital search tree holding 1(001), 4(100), 2(010), and 5 (101) inserted in that order.

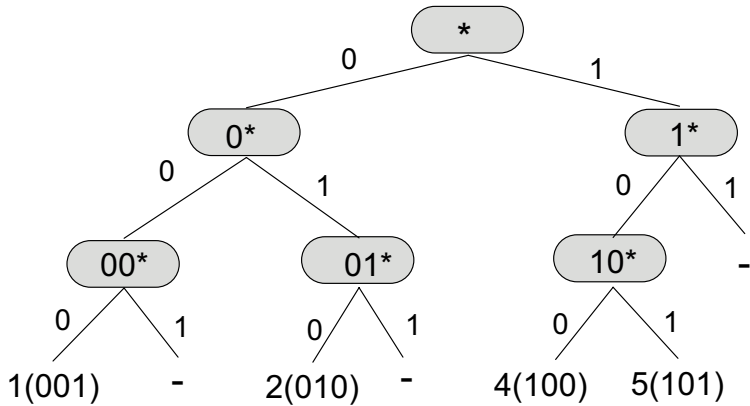


Figure 39.2
An example trie for the same collection as shown in the digital search tree of Figure 39.1. A “-” is used to represent a reference that is *null*.

39.4 Selecting a Data Structure

All data structures used for implementing `DigitizedOrderedCollection` are some form of a **trie** (derived from the word **retrieval** and pronounced “try”). To illustrate the similarities between tries and an abstract search tree (see Chapter 31), we briefly discuss digital search trees.

A **digital search tree** is a type of an abstract search tree in which each tree node can have up to b children where b is the base of the digitizer. The choice of which way to branch at each internal node is not made by a comparison. Instead for a node at level ℓ , the branch is selected based on the value of digit ℓ of the element held in that tree node. Like radix sort, the advantage of a digital search tree is that it is not comparison based. However, an **inorder traversal** may not visit the elements in lexicographical order. Recall that an inorder traversal is a recursive procedure that begins at the root. It recursively visits the left subtree (if it is not *null*), then visits the root, and then recursively visits the right subtree (if it is not *null*). The order defined by an inorder traversal is the order in which the elements are visited.

As a simple example, consider using the binary representation of the following numbers: 1 (001), 2 (010), 4 (100), and 5 (101). The digital search tree that results when inserting these elements in the order 1, 4, 2, 5, is shown in Figure 39.1. At the root, the search branches left if the leftmost bit is a 0, and branches right if the rightmost bit is a 1. At the next level, the search proceeds left if the middle bit is 0, and right if the middle bit is 1. Finally, at the lowest level, the search branches left if

the rightmost bit is 0 and left if the rightmost bit is 1. Observe that performing an inorder traversal would yield 2, 1, 5, 4.

A trie can be viewed as a variation of a digital search tree in which the elements are held only at the leaf level. A trie holding 1, 2, 4, and 5 is shown in Figure 39.2. A “-” is used to represent a *null* child. Unlike a digital search tree, the structure of a trie is independent of the order in which the elements are inserted. As with a B+-plus tree (Chapter 37) the leaves are chained in a doubly linked list to provide constant time support for moving forward and backward in the iteration order. Observe that the leaves appear in sorted order within an inorder traversal of the trie. Another important property of a trie, is that each internal node x can be labeled with a prefix such that an element is in the subtree rooted at x if and only if it has the prefix associated with x . This property enables a trie to efficiently support methods that find all elements in the collection that have a given prefix, or to find the element(s) in the collection that have a longest common prefix with a given element.

All trie data structures require that no element in the collection is a prefix of another element in the collection. One way to enforce this requirement is with a prefix free digitizer. However, this is not always necessary. For example, IP addresses are naturally prefix free because they are unique and all of the same length. In addition, the method to insert new elements into the collection must ensure that the collection does not contain duplicates.

In selecting among the trie data structures, there is a trade-off between the simplicity of the trie and the space usage. The Trie is the simplest, but unless there are usually at least two distinct elements for each occurring prefix, it is very inefficient in its use of space. A compact trie reduces the space usage of a trie by performing compaction at the leaf level. A compressed trie includes further compaction. Another way to reduce the space usage is with a ternary search trie that uses three-way branches as opposed to b -way branching where b is the base for the digitizer. When $b = 2$ and the elements in the collection are naturally prefix free, a Patricia trie further reduces space usage by having each trie node serves as both an internal node and a leaf node. Morrison [118] selected Patricia as an acronym for “practical algorithms to retrieve information coded in alphanumeric.”

This chapter concludes with a brief examination of some additional variations that can improve performance and reduce space usage. We also discuss **suffix trees** that allow efficient search for any sequence of digits and **indexing tries** that reduce the space complexity when indexing by storing the index within the document as a substitute for the element itself.

39.5 Terminology

We use the following definitions throughout our discussion of data structures for a digitized ordered collection.

- We say a collection C is **prefix free** if for any two distinct elements e_1 and e_2 in C , e_1 is not a prefix of e_2 . By definition, an element is a prefix of itself. Thus, if C is prefix free it cannot contain two equivalent elements.
- A **trie node** refers to either an internal node or leaf node of a trie.
- A **trie leaf node** refers to a leaf node of the trie.
- The **ordered leaf chain** is a doubly linked list holding all trie leaf nodes in the iteration order, which by definition is in sorted order with respect to the provided digitizer.
- For a trie node x , we let $T(x)$ denote the set containing all of the elements in the subtree rooted at x .

- The **level** of a node x in a rooted tree is defined recursively as:

$$level(x) = \begin{cases} 0 & \text{if } x \text{ is the root} \\ level(x.parent) + 1 & \text{otherwise.} \end{cases}$$

- For a trie node x that is the i^{th} child of parent p , we define the **left siblings** of x to be $\{p.child(j) \mid 0 \leq j < i \text{ and } child(j) \neq null\}$.
- For a trie node x that is the i^{th} child of parent p , we define the **right siblings** of x to be $\{p.child(j) \mid i < j < b \text{ and } child(j) \neq null\}$, where b is the base for the digits.
- For node x , the **branch position**, $x.bp$, is the digit position used to determine which branch to follow in a search. For example if a search for feed# is at node x , a branch position of 0 uses the “f,” a branch position of 1 uses the “e,” and so on.
- We define the **child index** for element e at node x , $childIndex(e,x)$, as the value returned by $digitizer.getDigit(e,x.bp)$. That is, the child index for element e is the index for the next node on the search path defined by e . For example, with $e = \text{feed\#}$, $x.bp = 0$, and when using the default digitizer (see [page 620](#)), $childIndex(e,x)$ is 6 since “f” is the 6th letter in the alphabet.
- We define the **associated child** of element e at node x as the child at index $childIndex(e,x)$. That is, the associated child index for element e is the next node on the search path defined by e .
- We say that a collection is **dense** if most of the possible elements that differ by a single digit from an element in the collection are also contained within the collection.

39.6 Competing ADTs

A digitized ordered collection can be used only when all elements in the collection can be viewed as a sequence of digits and a digitizer can be defined so that $a \prec b$ exactly when the application considers a to be smaller than b . If such a requirement cannot be satisfied, then another ADT is needed. Also, if the *getDigit* method of the digitizer is expensive, yet there is an efficient method to compare two elements in the collection, then another ADT should be considered since all data structures that implement a digitized ordered collection make heavy use of the *getDigit* method instead of making direct comparisons. We now briefly discuss other ADTs that may be appropriate in a situation when a digitized ordered collection is being considered.

OrderedCollection ADT: An ordered collection is more appropriate than a digitized ordered collection when (1) the cost of performing $O(\log n)$ comparisons between elements is less than the average number of digits in an element, and (2) the application requires neither a method to efficiently find all elements in the collection that begin with a given prefix, nor a method to find the set of elements in the collection that have a longest prefix in common with a given element. Note that data structures that implement an ordered collection are generally more space efficient than data structures that implement a digitized ordered collection.

TaggedDigitizedOrderedCollection ADT: If it is more natural for the application to define a digitizer based on a tag associated with the element (versus by defining a digitizer over the elements held in the collection) and it is uncommon for multiple elements to have the same tag, then a digitized ordered tagged collection is more appropriate.

TaggedBucketDigitizedOrderedCollection ADT: If it is more natural for the application to define a digitizer based on a tag associated with the element and it is common for multiple elements to have the same tag, then a digitized ordered tagged collection is more appropriate.

Set ADT: If a digitized ordered collection is used to implement a symbol table (or some other set or mapping) in which the primary methods are to just insert elements, search for elements, and remove elements, then the Set ADT should be considered. These are particularly favorable if a constant number of comparisons between elements is less expensive than performing *getDigit* for the number of digits that must typically be examined to distinguish the desired elements from all other elements in the collection. A set should also be considered if minimizing space usage is important.

39.7 Summary of Digitized Ordered Collection Data Structures

This section summarizes the strengths and weaknesses of our *DigitizedOrderedCollection* implementations in terms of application requirements. Table 39.3 provides a visual summary of these trade-offs. The methods that are trivially implemented in constant time for all ordered collections are not shown. Figure 39.4 shows an example instance of a trie, compact trie, and compressed trie holding the same collection.

Trie: The trie data structure is the simplest *DigitizedOrderedCollection* implementation. However, it typically requires significantly more space than the other data structures. It should be used only when most prefixes p that occur have at least two distinct elements with that prefix. The search path to any element is exactly the number of digits (including the end of string character) in the element.

CompactTrie: The compact trie data structure modifies the trie by replacing any leaf that has no siblings by its parent. A compact trie reduces the space complexity of a trie without any additional cost except that the search, insert, and remove methods are slightly more complicated. The length of the search path in a compact trie is at most the number of digits in the elements. More specifically, it is the length of the prefix needed to distinguish the given element from all other elements in the collection.

CompressedTrie: The compressed trie performs additional compression on a compact trie. While the compact trie ensures that all leaves have at least one sibling, there can be an internal node with a single child. A compressed trie ensures that all nodes have at least two children by replacing *any* node with a single child by that child. As a consequence, the level of a node in a compressed trie is not sufficient to determine the branch position. Thus, an instance variable is added to each internal node to store its branch position.

In the trie and compact trie, the common prefix for all descendants of an internal node is defined by the path to reach that node. In a compressed trie it is necessary to store this prefix, which can be done by keeping a reference to an arbitrary leaf in its subtree. Because these additional instance variables are needed, the compact trie could use less space than the compressed trie for a very dense collection. The length of the search path in a compressed trie is at most the length of the prefix needed to distinguish the given element from all other elements in the collection. However, in general it will be shorter.

For most common uses (such as indexing documents or web pages), the compressed trie will reduce the space usage without any cost in time complexity. Figure 39.4 shows a trie, compact

Key						
		○	○	⊖	●	•
		Excellent	Very Good	Good	Fair	Method does nothing
Method		Trie	CompactTrie	CompressedTrie	PatriciaTrie	TernarySearchTrie
Digitized Ordered Collection Methods	add(e)	○	○	○	○	⊖
	addAll(c), per element in c	○	○	○	○	⊖
	addTracked(e)	○	○	○	○	⊖
	clear(), per element	⊖	⊖	⊖	⊖	⊖
	completions(prefix) *	○	○	○	○	⊖
	contains(e)	○	○	○	○	⊖
	ensureCapacity()	•	•	•	•	•
	getEquivalentElement(e), getLocator(e)	○	○	○	○	⊖
	iterator(), iteratorAtEnd()	⊖	⊖	⊖	⊖	⊖
	longestPrefixMatches(prefix) *	○	○	○	○	⊖
	max(), min()	○	○	○	○	⊖
	predecessor(e)	○	○	○	○	⊖
Other Issues	remove(e)	○	○	○	○	⊖
	retainAll(c), per element in c	○	○	○	○	⊖
	successor(e)	○	○	○	○	⊖
	accept(v), toArray(), toString(), per element	⊖	⊖	⊖	⊖	⊖
Locator Methods	trimToSize()	•	•	•	•	•
	typical space	●	⊖	⊖	⊖	○
	requires b=2 (cannot add end of string character)				✓	
Locator Methods	structure uniquely defined by collection	✓	✓	✓		
	advance(), get(), hasNext(), next()	⊖	⊖	⊖	⊖	⊖
	remove()	○	○	○	○	⊖
	retreat()	⊖	⊖	⊖	⊖	⊖

Time Complexity

- ⊖ O(1) time
- O(d) or O(d_{max}) time
- ⊖ O(d log b) expected time
- O(n) time
- * additional O(|x|) ime where x is returned

Expected Space Usage

- ⊖ ≈ 2n (only applies when b = 2)
- ≈ 3n(1 + 1/ln b) ≈ 4n when b = 27
- ⊖ ≈ bn / ln b ≈ 8n when b = 27
- > bn(1 + 1/ln b) ≈ 35n when b = 27

Table 39.3 Trade-offs among the data structures that implement the *DigitizedOrderedCollection* interface.

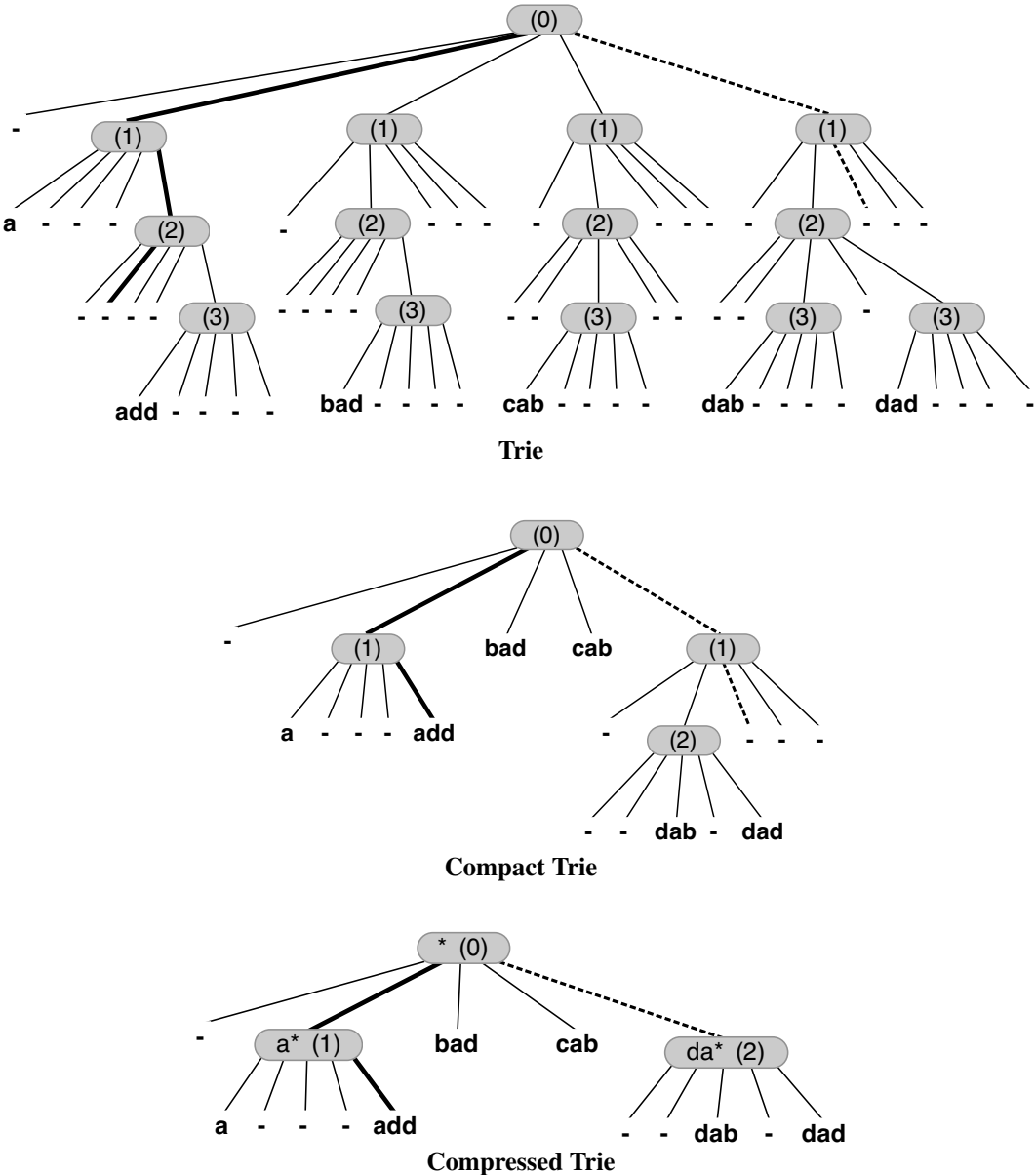


Figure 39.4
An example trie, compact trie, and compressed trie holding the elements `a#`, `add#`, `bad#`, `cab#`, `dab#`, and `dad#`, where `#` is the end of string character. The children of each node from left to right correspond to the characters `a`, `b`, `c`, `d`. A dash represents a *null* child. Each internal node is labeled with its branch position. Compressed trie nodes also show the common prefix for all descendants. Shown in bold in each trie is the search path for an unsuccessful search for `ada#`. The dashed path is an unsuccessful search for `db#`.

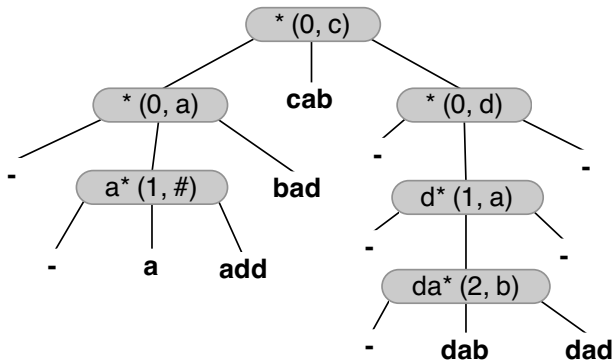


Figure 39.5

An example ternary search trie for the same collection as shown in the tries of Figure 39.4 when inserted in the order cab#, add#, dad#, dab#, bad#. Each internal node shows the common prefix followed by an ordered pair with the branch position and the digit used for branching.

trie and compressed trie holding the same collection. Also, at the start of the chapter for each implementation, we show the representation of the digitized ordered collection $C = \{\text{ad\#}, \text{add\#}, \text{bad\#}, \text{cab\#}, \text{cafe\#}, \text{dab\#}, \text{deed\#}, \text{fee\#}, \text{and feed\#}\}$. For, this example the trie has 21 internal nodes, the compact trie has 8 internal nodes, and the compressed trie has only 5 internal nodes.

PatriciaTrie: The Patricia trie is a variation of a compressed trie that can be used when the digitizer has base 2 and the collection is naturally prefix-free (without adding an end of string character). The most common application is for fixed-length binary strings (such as ASCII codes or IP addresses). A Patricia trie reduces the space usage of the compressed trie by letting each node serve the role of both an internal node and leaf node. The space savings can be further improved with an **indexing trie** as discussed in Section 39.10.

TernarySearchTrie: The ternary search trie (often referred to as a **TST**) is a hybrid between a trie and a binary search tree that combines the time efficiency of a trie with the space efficiency of a binary search tree (Chapter 32). In a ternary search trie, each node has three children. The branch to take during search is determined by both a branch position and the digit in that position for a distinguished descendant leaf node. Elements with a smaller digit use the left branch, elements with a equal digit use the middle branch, and elements with a larger digit use the right branch. A ternary search trie holding the same collection as in Figure 39.4 is shown in Figure 39.5. The choice of the distinguished leaf node for each internal node depends on the order in which the elements are inserted. So unlike a trie, compact trie, and compressed trie, whose structures are a function of the elements in the collection, the structure of a ternary search trie is not uniquely determined by the elements in the collection, and can be unbalanced if the elements are not inserted in a near random order.

With the trie, compact trie, and compressed trie, the branch position increases by at least one with each step taken along the search path. With a ternary search trie, the branch position only changes when the middle child (*child(1)*) is followed. As a result, the search path for an element can have a length greater than the number of digits in the element.

Figure 39.6 shows the class hierarchy for the data structures presented in this book for the DigitizedOrderedCollection, and Figure 39.7 shows the relationship among the trie node interfaces and classes used in our implementations.

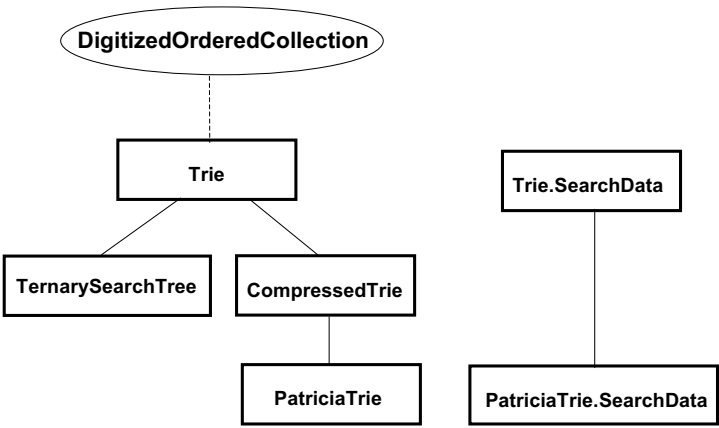


Figure 39.6
The class hierarchy for the Digitized OrderedCollection classes.

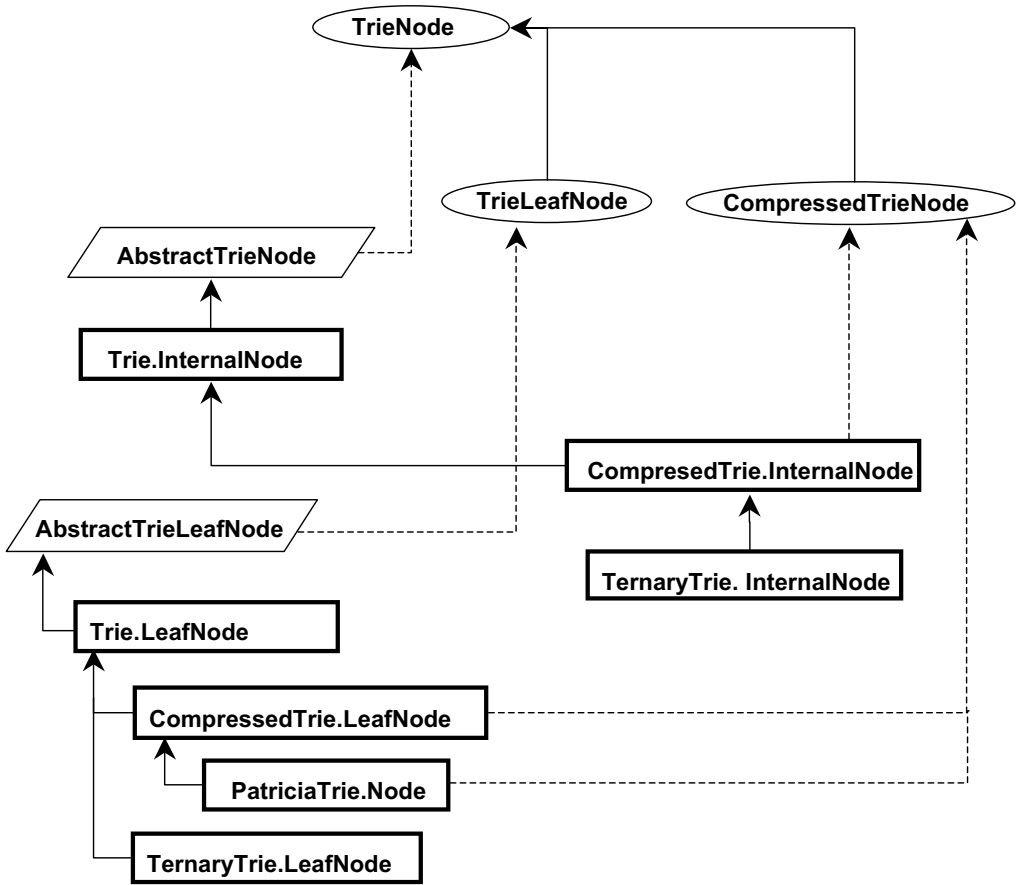


Figure 39.7
The class hierarchy for trie node classes used within the digitized ordered collection data structures.

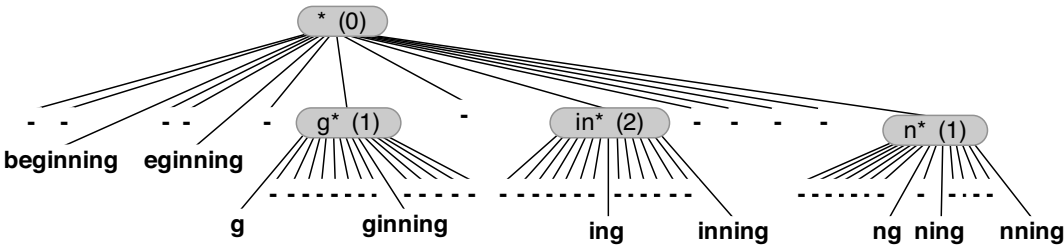


Table 39.8 A suffix tree for the word “beginning” using a compressed trie.

39.8 Trie Variations

In our trie implementation, an array of b elements holds the child references. A variation designed to reduce the space complexity of the trie is the **list trie**, where a linked list holds the child references. Knuth [97] shows that the expected number of pointers in a randomly generated list trie (including the child and sibling pointers) is roughly $2n \left(1 + \frac{1}{\ln b}\right)$, and Clément et al. [40] report that the time complexity for search is roughly 3 times slower than that of a ternary search trie. In contrast, the space complexity of a list trie is roughly 67% of that for a ternary search trie.

Knuth [97] discusses a variation of a compact trie in which any subtree with b or fewer elements is replaced by a list. This modification reduces the expected number of child references from $\frac{b}{\ln b} n$ to $\frac{n}{\ln b}$ with only a small increase in the expected search cost. However, care would be required to handle insertions and deletions.

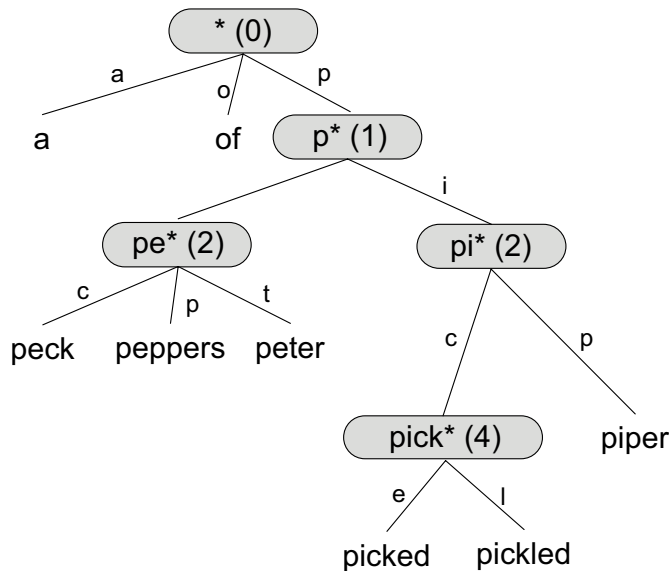
Bentley and Sedgwick [25] propose a variation of a ternary search trie where the root is replaced by any array of size b^2 where b is the base. That is, the root branches based on the first two digits, and the remainder of the trie is a ternary search trie. They call this a TST with b^2 branching at the root. In practice, such a trie benefits from the space savings of a ternary search trie while not suffering the increased search cost. Sedgwick [135] discusses this variation in slightly more depth.

39.9 Suffix Trees

A common application for a digitized ordered collection is searching for any substring within a document. While a trie can efficiently find any prefix, it cannot search for an arbitrary sequence in the middle of a document. A **suffix tree** is a trie that holds all suffixes (or suffixes of elements) from a document. Figure 39.8 shows an example suffix tree. Observe that any digitized ordered collection data structure could be used for the suffix Tree.

Since a digitized ordered collection allows an efficient search for all elements with a given prefix, and all suffixes are stored in the suffix tree, a suffix tree supports efficiently finding the occurrence of any substring in a document. For example, suppose one wanted to search to see if “inn” occurs in “beginning.” Observe that a search for inn# in the suffix tree of Figure 39.8 can find that inn# is a prefix of the leaf labeled with “inning.”

When using a suffix tree for indexing a document, typically a **tagged digitized ordered collection** would be used where the digitized element (corresponding to a suffix) is the tag, and the associated data is a list of all indices in the document where the given element begins.

**Figure 39.9**

A compressed trie holding the words in “Peter Piper Picked a Peck of Pickled Peppers.” Null children are omitted. Edges are labeled with the corresponding digit.

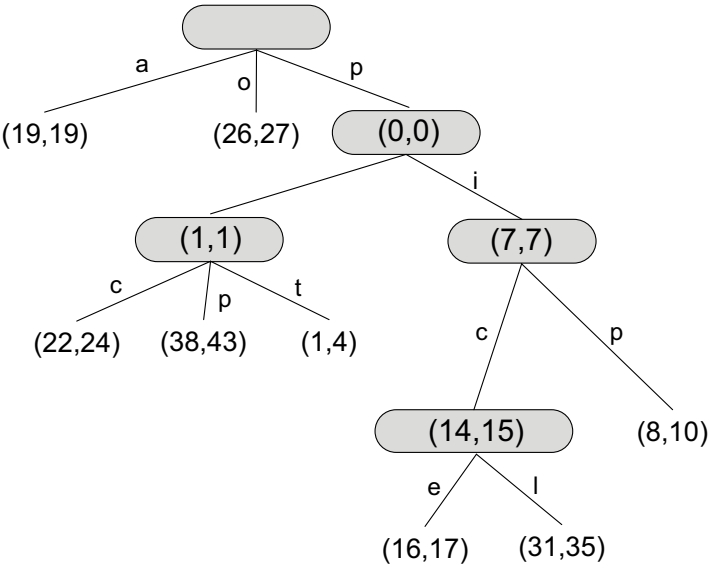
39.10 Indexing Tries

When indexing the words in a stored document, additional space savings over a compressed trie can be obtained by letting each internal node and leaf node store not the words themselves, but instead the start index of the word in the document and the number of characters. For example, consider the document “Peter Piper Picked a Peck of Pickled Peppers.” A compressed trie holding each of the words in this document is shown in Figure 39.9.

Instead of having each internal node x include instance variables for the branch position and a reference to a leaf node that is a descendant of x , instead it holds a starting and ending index into the document for the characters in common in the shared prefix of all descendants of x that were not in common for all descendants of x ’s parent. Similarly, instead of each leaf node holding a reference to a digitized element, instead it holds a beginning and ending index in the document for the element. This is often called an **indexing trie**. Figure 39.10 shows a sample indexing trie corresponding to the compressed trie in Figure 39.9.

Design Notes: The trie implementations illustrate several design patterns.

- Each child for a trie node could be an internal node (which includes the array of child references) or a leaf node (which includes the next and previous pointer for the ordered leaf chain). Since the internal node and leaf node each have instance variables not needed by the other, defining either as a subclass of the other would waste space. Thus, we use internal and leaf node interfaces that allow each child reference to be either an internal node or a leaf node. Moreover, we introduce the *TrieNode* interface that all trie nodes implement, and we let the children of every node be declared of this general type. Finally, in each trie implementation, we use the factory design pattern to create the appropriate internal node and leaf node types. Figure 39.7 shows the relationships among the trie node interfaces and classes.



Document:

index in document	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	2	
document	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
	p	e	t	e	r		p	i	p	e	r		p	i	c	k	e	d		a	

index in document	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	3	4	4	4	4
document	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3
	p	e	c	k		o	f		p	i	c	k	l	e	d		p	e	p	p	e	r	s

Figure 39.10
An indexing trie holding the same collection as that of Figure 39.9. The table above the figure shows the index within the document for each character.

- Like many other data structure implementations, we introduce an internal *find* method that is used by many public methods. However, there are several variables that must be returned (e.g., the trie node where the search ends and the level of that node). Since Java does not support call-by-reference, some mechanism is needed. Our B-tree implementation (Chapter 36) introduces a global variable to save information from the internal search. However, the trie implementations often need to save several variables computing in *find*. We introduce the SearchData class (Section 41.5) to package these variables. However, we want to avoid creating a new object (which then becomes garbage) with every call to *find*. Our solution is to use an object pool (as described in Section 5.5) to reuse the return objects. It is important that every *allocate* method made into the pool is paired with a *release* method that occurs in the same method. Otherwise, the pool will grow unnecessarily large. We initialize *pool* to be an initially empty pool of SearchData objects. Section 41.5 defines the SearchData class.
- We use the factory method design pattern to enable method reuse. Often the only change required by a subclass is that some instances allocated by the method need to have a more specific type. The factory method provides a level of indirection in calling the constructor for these classes. We can override the factory method for each subclass to create specific node types throughout the code, without the need to override the methods that call the factory to create those nodes.

- We use the strategy design pattern to decouple the definition of the digitizer from the data structure implementation.

39.11 Further Reading

Coffman and Eve [41] first proposed the digital search tree. Fredkin [60] introduced the name “trie.” Tries and Patricia tries were introduced by Morrison [118]. Trie compaction was studied by Al-Suwaiyel and Horowitz [8].

The analysis of the expected search time for digital search trees and tries is presented by Knuth [97]. Clément et al. [40] provide detailed analysis of various implementations of tries including the ternary search trie. Sedgewick [135] and Clément et al. [40] present empirical results comparing different trie implementations including the TST (ternary search trie) variation in which the root branches based on the first two digits, yielding b^2 branching at the root. Further information can be found in the books of Horowitz et al. [86], Sedgewick [135], and Goodrich and Tamassia [76].

Chapter 40

Trie Node Types

package collection.ordered.digitized

In this chapter we present the interfaces and abstract class definitions for the trie node and trie leaf node. One possible implementation (as illustrated in Chapter 37 for the B+-Tree) would be to have the leaf node extend the internal node. Such an implementation would waste space since leaf nodes would inherit instance variables (such as the array of child references) that are not used. The interfaces and classes provided in this chapter provide the foundation for any tree-based data structure in which the leaf nodes are of a different type than the internal nodes. However, for the PatriciaTrie, the desired behavior is to have the node class extend both the internal and leaf node classes.

Having polymorphic use of nodes would provide a solution to this problem. While Java does not support **multiple inheritance** meaning that one class extends two others, Java does allow one class to implement multiple interfaces. Therefore, to allow a Patricia node to serve both as an internal node and a leaf node, we define the PatriciaTrieNode to implement both the TrieNode and TrieLeafNode interfaces.

40.1 Trie Node Interface

The following methods are included in the *TrieNode* interface.

***TrieNode*<E> child(i):** Returns a reference to the i^{th} child.

E data(): Returns the data (if any) associated with this trie node. All data elements are held in leaf nodes, but for some trie implementations the internal nodes hold a reference to an element that begins with the common prefix shared by all of its descendants.

boolean isLeaf(): Returns *true* if and only if this trie node is a leaf.

***TrieNode*<E> parent():** Returns a reference to the parent (or *null* for the root).

***void setParent(TrieNode*<E>):** Sets the parent reference to be the given trie node.

40.2 Abstract Trie Node Class

***AbstractTrieNode*<E>** implements *TrieNode*<E>

Since many of the method implementations for TrieNode are shared by several data structures, we provide an AbstractTrieNode class.

Internal Representation: Each abstract trie node includes a parent reference.

```
TrieNode<E> parent;
```

The *isLeaf* method by default treats each node as an internal node. Thus it returns *false*.

```
public boolean isLeaf() {
    return false;
}
```

The *parent* method returns a reference to the parent trie node.

```
public TrieNode<E> parent() {
    return parent;
}
```

The *setParent* method takes *parent*, a reference to the new parent reference.

```
public void setParent(TrieNode<E> parent) {
    this.parent = parent;
}
```

The *child* method takes *i*, the index for the desired child. The abstract method returns *null*. It must be overridden when the trie node has children.

```
public TrieNode<E> child(int i) {
    return null;
}
```

Finally, the *data* method returns the data (if any) associated with the trie node. The abstract method returns *null*. It method must be overridden when the trie node has associated data.

```
public E data() {
    return null;
}
```

40.3 Trie Leaf Node Interface

TrieNode<E>

↑ *TrieLeafNode*<E>

The following methods are included in the *TrieLeafNode* interface that extends the *TrieNode* interface. Most of the methods of this interface support the construction and use of the ordered leaf chain.

void addAfter(TrieLeafNode<E> ptr): Inserts this trie leaf node into the ordered leaf chain immediately after the trie leaf node referenced by *ptr*.

boolean isDeleted(): Returns *true* if and only if this trie leaf node is not in use. By definition we say that FORE and AFT are deleted since they do not hold an element in the collection.

void markDeleted(): Marks this trie leaf node as no longer in use.

***TrieLeafNode*<E> next():** Returns a reference to the next leaf node in the ordered leaf chain.

***TrieLeafNode*<E> prev():** Returns a reference to previous leaf node in the ordered leaf chain.

void remove(): Removes this trie leaf node from the ordered leaf chain.

***void setNext(TrieLeafNode*<E> *newNode*):** Sets the next element in the ordered leaf chain to the leaf node referenced by *newNode*.

***void setPrev(TrieLeafNode*<E> *prevNode*):** Sets the previous element in the ordered leaf chain to *prevNode*.

40.4 Abstract Trie Leaf Node Class

AbstractTrieLeafNode<E> implements ***TrieLeafNode***<E>

To enable all trie-based data structure to share methods that maintain the ordered leaf chain and the redirect chain when an element is removed so that a tracker to it can still be used for iteration, we include them in this **AbstractTrieLeafNode** class.

Internal Representation: The **AbstractTrieLeafNode** class has a reference to the next and previous elements in the ordered leaf chain.

TrieLeafNode <E> next;	<i>//refers to next node in ordered leaf chain</i>
TrieLeafNode <E> prev;	<i>//refers to previous node in ordered leaf chain</i>

The *isLeaf* method for the **AbstractTrieLeafNode** returns *true*.

```
public boolean isLeaf() {
    return true;
}
```

The *next* method returns a reference to the next leaf node in the ordered leaf chain.

```
public TrieLeafNode<E> next() {
    return next;
}
```

The *next* method returns a reference to the previous leaf node in the ordered leaf chain.

```
public TrieLeafNode<E> prev() {
    return prev;
}
```

The *setNext* method takes *nextNode*, the value to set the next pointer.

```
public void setNext(TrieLeafNode<E> nextNode) {
    next = (TrieLeafNode<E>) nextNode;
}
```

The *setPrev* method takes *prevNode*, the value to set the previous pointer.

```
public void setPrev(TrieLeafNode<E> prevNode) {
    prev = (TrieLeafNode<E>) prevNode;
}
```

The *markDeleted* method marks this trie leaf node as no longer being in the collection. We will use a *prev* pointer of *null* to indicate that a leaf node is not in use.

```
public void markDeleted() {
    prev = null;
}
```

The *isDeleted* method returns *true* if and only if this trie leaf node is not in use.

```
public boolean isDeleted(){
    return prev == null;
}
```

The *addAfter* method takes *ptr*, and places this trie leaf node after the one referenced by *ptr* in the ordered leaf chain.

```
public void addAfter(TrieLeafNode<E> ptr){
    this.setNext(ptr.next());
    ptr.setNext(this);
    this.setPrev(ptr);
    next.setPrev(this);
}
```

The *spliceOut* method removes this leaf node from the ordered leaf chain and marks the removed trie leaf node as deleted.

```
public void remove(){
    prev.setNext(next);           //removes from ordered leaf chain
    next.setPrev(prev);
    markDeleted();                //marks as deleted
}
```

Chapter 41

Trie Data Structure

package collection.ordered.digitized

AbstractCollection<E> implements *Collection*<E>

↳ **Trie**<E> implements *DigitizedOrderedCollection*<E>, *Tracked*<E>

Uses: Java array and references

Used By: TaggedTrie (Section 49.10.2)

Strengths: The search path to any element is exactly the number of digits (including the end of string character) in the element. Thus, no comparisons are made when searching for an element.

Weaknesses: A trie generally requires significantly more space than the other *DigitizedOrderedCollection* data structures, except when the collection is **dense** meaning that most elements share a relatively long common prefix with another element in the collection.

Critical Mutators: none

Competing Data Structures: Unless the collection is very dense, a more space efficient implementation such as the compact trie (Chapter 42) or compressed trie (Chapter 43) should be considered. If space efficiency is more important than time efficiency, then a ternary search trie (Chapter 45) is a viable option. Finally, if the digitizer is base 2 and the collection is naturally prefix-free, then a Patricia trie (Chapter 44) should be considered.

41.1 Internal Representation

Instance Variables and Constants: The following instance variables and constants are defined for the Trie. We introduce the constant *NO_CHILD* to return as an index for a non-existent child. The trie leaf node *FORE* is logically just before the first element in the collection and also serves as the head sentinel for the ordered leaf chain. Similarly, the trie leaf node *AFT* is logically just after the last element in the collection and serves as the tail sentinel for the ordered leaf chain.

```
static final int NO_CHILD = -1;
final TrieLeafNode<E> FORE = new LeafNode(null);
final TrieLeafNode<E> AFT = new LeafNode(null);
```

We also introduce *root*, a reference to the root of the trie, *digitizer*, the digitizer for the trie, *childCapacity*, the maximum number of children that a node can have, and *base*, the base used by the digitizer. Every digit must be mapped to an integer from 0, . . . , *base-1* where the digit that maps to 0 is lexicographically first, the digit that maps to 1 is lexicographically second, and so on.

```

TrieNode<E> root;
int childCapacity;           //maximum number of children a trie node can have
Digitizer<? super E> digitizer;
int base;                     //the digitizer's getDigit method returns int in {0, ..., base-1}

```

The internal *find* method is used by all of the trie data structures to search for a desired element, or the position where an element falls in the iteration order. However, *find* needs to return several items including the reference to a trie node and the level of that node. Since Java does not support call-by-reference, there are two approaches that avoid creating a garbage object with each call to the internal find method. Our B-tree implementation (Chapter 36) used a global variable to hold the information needed beyond the *find* return value. Here, we use an object pool (as described in Section 5.5) to reuse return objects whose variables hold the required return information. It is important that every *allocate* method made into the pool is paired with a *release* method that occurs in the same method. Otherwise, the pool will grow unnecessarily large. We initialize *pool* to be an initially empty pool of SearchData objects. Section 41.5 defines the SearchData class.

```

final Pool<SearchData> pool = new Pool<SearchData>() {
    protected SearchData create() {
        return createSearchData();
    }
};

```

Populated Example: Figures 41.1 shows a populated example of a trie. The bold path is the search path for *cafe#* and the dashed path is the search path for *deaf#*.

Terminology: We use the following terminology that is similar to that defined for Abstract-SearchTree (Chapter 31). Some of these definitions relate to *sd*, a SearchData instance. Among other things, each SearchData instance includes *ptr*, a reference to a trie node that defines its location.

- For trie node *x* we let *x.bp* denote the **branch position** for *x*. Illustrations show the branch position of each internal node in parentheses.
- We use ϵ to denote the empty string.
- We define the **search prefix** of trie node *x*, *sp(x)* recursively as follows:

$$sp(x) = \begin{cases} \epsilon & \text{if } x = \text{root} \\ sp(x.parent) + d_x & \text{otherwise} \end{cases}$$

where $+$ is the concatenation operator, and $d_x = \text{digitizer.getDigit}(x.data, x.bp)$. For example, if *x* is the internal node with branch position 2 that is the ancestor of *dab#* and *dad#* in the Trie of Figure 41.1, $sp(x) = \epsilon + \text{"d"} + \text{"a"} = \text{"da"}$. Observe that all children of *x* begin with the prefix "da."

- For an internal trie node *x*, we let $T(x)$ denote the set of trie leaf nodes that are in the subtree rooted at *x*.
- A **descendant** of *x* is any node in $T(x)$, including *x*.
- An **ancestor** of *x* is any trie node reachable from *x* by following zero or more parent references from *x*, so *x* is an ancestor of itself.

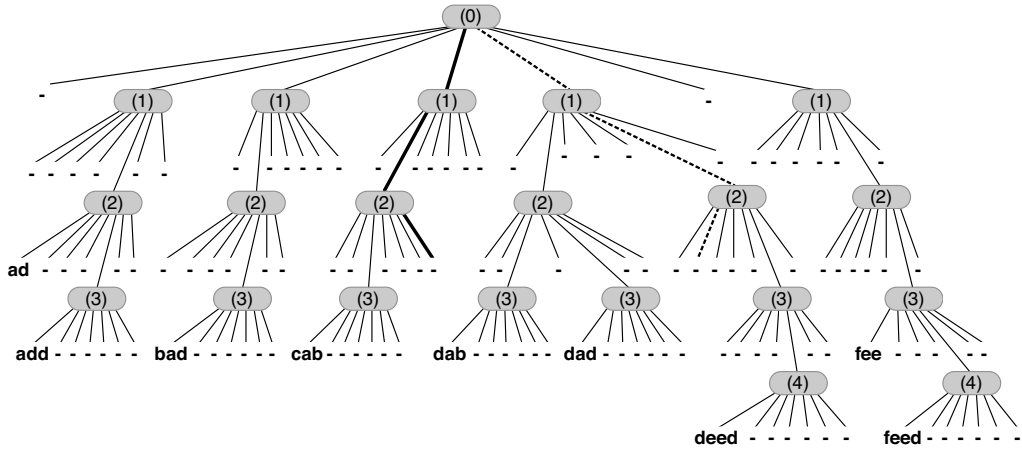


Figure 41.1

A populated example of a trie holding `ad#`, `add#`, `bad#`, `cab#`, `cafe#`, `dab#`, `dad#`, `deed#`, `fee#`, and `feed#`. The structure of the trie is the same, regardless of the order in which the elements are added. The leftmost branch from each node corresponds to the end of string character #. Each leaf is visualized by showing its associated data (without the end of string character). The search path for `cafe#` is shown in bold, and the search path for `deaf#` is shown as a dashed line.

- For a trie leaf node x , we say that x is **in the collection** if and only if $x \in T(\text{root})$.
- We let “ $\langle \rangle$ ” denote an empty sequence. Also, when applied to a sequence, “ $+$ ” denotes the concatenation operator. We use $\sum_{i=0}^s x_i$ denote $x_0 + \dots + x_s$.
- The iteration order given by $\text{seq}(\text{root})$ is defined recursively as

$$\text{For } x \text{ a trie node, } \text{seq}(x) = \begin{cases} \langle \rangle, & \text{if } x \text{ is null} \\ \langle x.\text{data} \rangle, & \text{if } x \text{ is a leaf node} \\ \sum_{i=0}^{b-1} x.\text{child}(i) & \text{otherwise} \end{cases}$$

where b is the base for the digitizer. Observe that $\text{seq}(\text{root})$ contains the elements in the collection from left to right.

- For a SearchData instance sd , we say that sd is **at node** x when $sd.\text{ptr} = x$. We also say that SearchData instance sd has **search location** x when $sd.\text{ptr} = x$.

Abstraction Function: Let T be a trie. The abstraction function is

$$AF(T) = \text{seq}(\text{root}).$$

Optimizations: Our implementation uses an array representation for the child references. The advantage of this approach is that it takes constant time to access the i^{th} child of a node. The drawback is that every node requires an array of size b even if most of its children are *null*. A **list trie**, which replaces the array of child references by a list of the non-null child references, is an alternative that optimizes space usage, at the cost of increasing the search time.

There is a trade-off between the insertion time and support for fast iteration. In the provided implementation, the cost for insertion is increased slightly to maintain the ordered leaf chain. An alternative approach would be similar to that used for the binary search tree (Chapter 32) in which

no leaf chain is maintained but rather the previous and next element in the collection can be computed as needed. Another option would be to use a visiting iterator as illustrated in the leftist heap (Chapter 26).

Finally, for ease of exposition, whenever the predecessor or successor is sought, a method is used to find the predecessor and then the ordered leaf chain is used to reach the successor. If desired, one could locally determine based on the search data location whether the predecessor or successor would be more efficient to find. Such an optimization could remove the need to move up the trie and then back to the bottom when inserting an element, finding the predecessor, or finding the successor.

41.2 Representation Properties

We inherit SIZE and introduce the following additional representation properties.

REACHABLE: The elements held within $T(\text{root})$ are exactly those in the collection.

PATH: For reachable trie node x , an element is in $T(x)$ if and only if it has the prefix $sp(x)$. Observe that this property implies that $sp(x)$ defines a unique search path leading to x .

LEAFPATH: For every leaf node $\ell \in T(\text{root})$, $\ell.\text{data}()$ returns $sp(\ell)$.

BRANCHPOSITION: The branch position for each internal trie node is the level of that trie node. That is, for an internal trie node x , $x.\text{bp} = \text{level}(x)$.

ORDEREDLEAFCHAIN: For a reachable trie leaf node, $x.\text{prev}$ references the trie leaf node holding the previous element in the iteration order, and $x.\text{next}$ references the trie leaf node holding the next element in the iteration order.

PARENT: For each internal trie node x , $x.\text{child}(i).\text{parent} = x$. Furthermore, $\text{root}.\text{parent} = \text{null}$.

NUMCHILDREN: The instance variable *numChildren* for trie node x is equal to the number of non-null children that x has. Also, every internal node has at least one child.

INUSE: For any node x , $x.\text{prev} = \text{null}$ if and only if $x \notin T(\text{root})$. That is, a trie node is in use exactly when $x.\text{prev} \neq \text{null}$.

REDIRECTCHAIN: When a tracked element is removed then the tracker is considered to be positioned just before the successor of the element at that time. If $x \notin T(\text{root})$, the chain of *next* references starting at x ends at node that x is considered to be just before.

41.3 Internal Node Inner Class

`AbstractTrieNode<E>` implements `TrieNode<E>`

↳ `Trie<E>.InternalNode` implements `TrieNode<E>`

The `InternalNode` inner class supports the `Trie` class. It extends the `AbstractTrieNode` class by adding two instance variables, *children* an array of b child references where b is the base for the

digitizer, and *numChildren*, the current number of non-null children for this internal node.

```
TrieNode<E>[] children;           //array of b child references
int numChildren;                 //number of non-null children
```

The constructor allocates an array with one slot for each possible child and sets the current number of children to 0. Recall that *childCapacity*, the maximum number of children a trie node can have, is set by the constructor to be the base of the digitizer.

```
InternalNode() {
    children = new TrieNode[childCapacity];
    this.numChildren = 0;
}
```

The *child* method takes *i*, the index for the desired child, and returns the i^{th} child. It throws an *IllegalArgumentException* when *i* is not between 0 and *childCapacity* − 1 (inclusive).

```
public TrieNode<E> child(int i){
    if (i < 0 || i ≥ childCapacity)
        throw new IllegalArgumentException();
    return children[i];
}
```

The *childIndex* method takes *element*, the element for which the index of the child is sought and *bp*, the branch position of the node on which this method is called. It returns the index for the next node on the search path defined by *element*.

```
public int childIndex(E element, int bp){
    return digitizer.getDigit(element, bp);
}
```

The *setChild* method takes *child*, the new child to add, *element*, the element defining the search path for the child, and *bp*, the branch position of the node on which this method is called. It sets the associated child for *element* to *child*. This method requires that *child* is not *null*. It returns the index at which *child* is placed.

```
protected int setChild(TrieNode<E> child, E element, int bp) {
    int i = childIndex(element, bp);
    if (children[i] == null)
        numChildren++;
    children[i] = child;
    child.setParent(this);
    return i;
}
```

Correctness Highlights: Whenever this method is used to add a child, incrementing *numChildren* maintains NUMCHILDREN. It also ensures that PARENT is preserved. ■

41.4 Leaf Node Inner Class

AbstractTrieLeafNode<E> implements *TrieLeafNode*<E>

└ **TrieNode.LeafNode** implements *TrieLeafNode*<E>

The *TrieLeafNode* is a simple inner class that supports many of the methods of the *Trie* class. The *TrieLeafNode* class extends the *AbstractTrieLeafNode* class by adding an instance variable *data* that holds the data element associated with the leaf.

E data;

The only methods that are added to the *AbstractTrieLeafNode* class are a constructor, an accessor for *data*, and the *toString* method.

```
LeafNode(E data) {this.data = data;}
```

```
public E data() {return data;}
```

41.5 Search Data Inner Class

All methods that involve searching for a given element in a trie use the internal *find* method to locate the desired element. Similarly, when inserting a new element into the trie, *find* is used to determine where the new element should be inserted. Recall that the branch position for a node is its level in the trie. Furthermore, the level is not stored in each node but is instead computed during the search. Thus, the *find* method must maintain both a reference to the current trie node and also the branch position of its current location. In addition, *find* returns the outcome of the search, including whether or not the target was found and, if not, information about the relationship of the target to the element in the trie node where the search ended. Section 41.6 describes the enumerated type *FindResult* used as the return type of *find*.

Each *SearchData* object *sd* contains *ptr*, a reference to the trie node that defines the location of *sd*, and *bp*, the branch position for *sd.ptr*. Recall that for the *Trie* class, if a leaf node is reached it is guaranteed to hold an element equivalent to the target. However, in the compact trie data structure, when the search ends at a leaf, the digits starting at the branch position of the leaf must be compared to see if the element held in the leaf is equivalent to the target. Thus, we include an instance variable *numMatches* to count the number of matches that occur after a search ends at a leaf. This instance variable is not used for the *Trie* class. While we could instead extend *SearchData* for *CompactTrie*, the only change would be the addition of this single instance variable. So, we instead choose to make *SearchData* general enough to support a compact trie.

```
TrieNode<E> ptr;           //references search data location
int bp;                   //branch position
int numMatches;           //for use by compact trie
```

To simplify later code that operates upon the variables held in a search data object, we provide some simple methods to support navigation and adjust variable values during search when moving to a child or parent. The *childBranchPosition* method takes *childIndex*, the index for the child of interest. It returns the branch position of that child.

```

int childBranchPosition(int childIndex) {
    return bp+1;
}

```

Correctness Highlights: By BRANCHPOSITION, as you move down one level in the tree, the branch position increases by one. ■

The *parentBranchPosition* method returns the branch position of the parent.

```

int parentBranchPosition() {
    return bp-1;
}

```

Correctness Highlights: By BRANCHPOSITION, as you move up one level in the tree, the branch position decreases by one. ■

The *atLeaf* method returns *true* if and only if the search data object is currently at a leaf node.

```

public boolean atLeaf() {
    return ptr.isLeaf();
}

```

The *atRoot* method returns *true* if and only if the search data object is currently at the root.

```

public boolean atRoot() {
    return ptr == root;
}

```

The *numMatches* method returns the number of digits of the element that have been matched so far.

```

public int numMatches() {
    return bp;
}

```

Correctness Highlights: By BRANCHPOSITION, the branch position is equal to the level of the current node. Since the level increases by one at each step in the search path, by PATH it follows that there is one match per level. Thus returning *branchPosition* yields the correct value. ■

The method *childIndex* takes *element*, an element to be considered on this search path, and returns the index of the child branch that would be followed to reach the given element from the current position of this search data object.

```

protected int childIndex(E element) {
    return ((InternalNode) ptr).childIndex(element, bp);
}

```

The *moveDown* method that takes *childIndex*, the index of the child to which to move, returns the given index if the SearchData instance moves successfully or *NO_CHILD* if the specified child does not exist.

```

protected int moveDown(int childIndex){
    TrieNode<E> child = ptr.child(childIndex);
    if (child == null)
        return NO_CHILD;
    bp = childBranchPosition(childIndex);
    ptr = child;
    return childIndex;
}

```

The *moveDown* method that takes *element*, the target, moves this SearchData instance down one level in the tree, and returns the child index for the current search location, or *NO_CHILD* if the SearchData instance did not move because a *null* child was encountered.

```

protected int moveDown(E element) {
    int childIndex = ((InternalNode) ptr).childIndex(element, bp);
    return moveDown(childIndex);
}

```

Correctness Highlights: Follows from PATH and the correctness of the TrieNode *childIndex* method, as well as the correctness of the *moveDown* method that takes the child index as a parameter. ■

The method *extendPath* is a convenience method that adds a new child node below the node at which the SearchData instance is positioned, and then moves down to the new child. It takes as parameters *element*, the element that defines the search path on which the child node should be placed, and *newChild*, the child node to be added. The method requires that this SearchData instance is currently positioned at an internal node.

```

protected int extendPath(E element, TrieNode<E> newChild){
    return moveDown(((InternalNode) ptr).setChild(newChild, element, bp));
}

```

The *moveUp* method moves up one level in the tree. It returns the branch position for current search location.

```

protected int moveUp() {
    bp = parentBranchPosition();
    ptr = ptr.parent();
    return bp;
}

```

Correctness Highlights: Follows by PARENT, the correctness of *parentBranchPosition*, and the correctness of the TrieNode *parent* method. ■

The *processedEndOfString* method takes *element*, the target. It returns *true* if and only if the last step of the search path processed the end of string character.

```

protected boolean processedEndOfString(E element) {
    return (digitizer.isPrefixFree() &&
        ptr != root && ptr.parent().child(0) == ptr);
}

```



```

    }
    if (atRoot())                                //Case 2: at root, and its not a left fork
        return;
    else
        moveUp();                                //Case 3: move up and continue looking for left fork
    }
}

```

Correctness Highlights: The while loop repeatedly checks to see if it has reached a left fork. If so, then it moves down to the left fork root and returns. Otherwise, it checks if the root has been reached in which case there is no left fork. If the search location is not the root, then the search moves up the trie by one level, and the process is repeated. We now discuss the correctness of each case.

Case 1: If *sd* is not at a leaf, this method must determine if *sd* has reached a left fork. (Observe that by definition of a left fork, if *sd* is at a leaf then it is not at a left fork.) The for loop considers the left siblings of the search location from the right to left. If *moveDown* is successful then the method is complete – the left fork root has been reached. Furthermore, if all left siblings have a *null* child, then the current search location is not a left fork.

Case 2: Upon reaching this step, it is known that the current search location is not a left fork. So if *sd* is at the root, there is no left fork on the path from *x* to the root, which implies that *x* is the first element in the iteration order.

Case 3: In this case, which is only reached when *sd* is not at a left fork and is not at the root, *sd* is moved to its parent, and the process is repeated.

The rest of the correctness follows from the order defined by the digitizer and LEAFPATH, as well as on the correctness of the *childIndex*, *moveDown*, and *moveUp* methods. ■

The *moveToMaxDescendant* method moves this search data object to the descendant of its current location that is last in the iteration order among its descendants.

```

public void moveToMaxDescendant() {
    while (!atLeaf()) {
        int i = childCapacity - 1;                //start at rightmost child
        while (moveDown(i) == NO_CHILD)           //moves to child i if it exist
            i--;
    }
}

```

Correctness Highlights: Follows by PATH and ORDEREDLEAFCHAIN. ■

By PATH, it follows that all elements that are descendants of node *x* are completions of the element *sp(x)*. Several public methods make use of the following recursive method that constructs a positional collection that holds all elements in $T(x)$ in the iteration order.

The *elementsInSubtree* method takes *c*, a collection to which all elements in the subtree rooted at this search location are added.

```

void elementsInSubtree(Collection<? super E> c){
    if (atLeaf())

```

```

        c.add(((TrieLeafNode<E>) ptr).data());
    else {
        for (int i = 0; i < childCapacity; i++)
            if (moveDown(i) != NO_CHILD) {
                elementsInSubtree(c);
                moveUp();
            }
    }
}

```

//recurse on child i

Correctness Highlights: Since a recursive call is made on each non-null child of the current search location, a call to *elementsInSubtree* will be made for all elements reachable from the original location. We now argue that they will be visited in sorted order by considering an arbitrary two descendants e_1 and e_2 where $e_1 \prec e_2$. By PATH, e_1 and e_2 share the same prefix $sp(x)$ where x is the current search location. Since $e_1 \prec e_2$, e_1 must have a smaller value for the digit at the branch position for x . Thus the recursive call is made for e_1 before e_2 , guaranteeing that e_1 precedes e_2 in the resulting positional collection. ■

41.6 FindResult Enumerated Type

Java supports an **enumerated type**, a class with a fixed number of instances. We introduce the enumerated type *FindResult* to capture the relationship between the target and the element at the end of a search. All Find Result values are named from the perspective of the target. We use *sd* to denote the SearchData instance.

For the Trie class, an unsuccessful search always ends at a *null* child. However, in the compact trie and compressed trie (and their subclasses), it is possible for the search for a given element to end at an internal node or a leaf node that is not equivalent to the target, as in the following two examples:

- Consider a search for *ada#* in the compact trie of Figure 39.4 of page 627. The search path is shown as a bold line. Observe that this search ends at the leaf (call it ℓ) holding *add#*. While it is known that the first two digits of *ada#* match *add#*, the remaining digits must be compared to determine if *ada#* is held in the collection. When they are compared, it is found that *ada#* is less than *add#*. So the search completes with *sd* at ℓ and returns *LESS*. This situation can occur in both a compact trie and compressed trie. Observe, that in this search no *null* child was reached.
- Consider a search for *db#* in the compressed trie of Figure 39.4. The search path is shown as a dashed line. The search ends at the internal node (call it x) labeled by “da* (2).” All elements in $T(x)$ have the prefix “da,” and $x.bp = 2$ which means that the next branch would look at digit 2 (where the leftmost digit is digit 0). However, first the “b” of *db#* must be compared with “a.” Based on this comparison, we can conclude that *db#* is greater than all elements in $T(x)$. So a search for *db#* will end with *sd* at x and return *GREATER*. This situation occurs only in a compressed trie. Again, observe that no *null* child was reached.

```

protected static enum FindResult
{EXTENSION, GREATER, LESS, MATCHED, PREFIX, UNMATCHED}

```


UNMATCHED: is returned when the search ends at a *null* child. In this situation, *sd* ends at the node from which the *null* child was reached.

PREFIX: is returned if *sd* is at a leaf ℓ for which the target is a prefix of $\ell.data$.

MATCHED: is returned if the *sd* is at a leaf node ℓ where $\ell.data$ is equivalent to the target.

EXTENSION: is returned if *sd* is at a leaf node ℓ for which the target is an extension of $\ell.data()$.

LESS: is returned if *sd* ends at an internal node or leaf node x (without reaching a *null* child) for which the target is less than $x.data()$, but not a prefix of it.

GREATER: is returned if *sd* ends at an internal node or leaf node x (without reaching a *null* child) for which the target is greater than $x.data()$, but not an extension of it.

For the Trie, LESS and GREATER, are never returned. However, these two return values are used for all classes that extend Trie.

41.7 Trie Methods

In this section, we present the methods for the Trie class.

41.7.1 Constructors and Factory Methods

The constructor takes *digitizer*, the digitizer to be used. It creates an empty trie that uses the given digitizer.

```
public Trie(Digitizer<? super E> digitizer) {
    super(Objects.DEFAULT_COMPARATOR);
    childCapacity = digitizer.getBase();
    this.digitizer = digitizer;
    root = null;
    FORE.setNext(AFT);
    AFT.setPrev(FORE);
}
```

Correctness Highlights: ORDEREDLEAFCHAIN is initially *true* since FORE is followed by AFT. The rest of the properties hold vacuously since there are no elements in the collection and $T(\text{root})$ is empty. ■

The *createSearchData* factory method returns a newly created and initialized search data instance. This method is called by the *allocate* method in the search data pool when there isn't a search data object available for reuse.

```
protected SearchData createSearchData() {
    return initSearchData(new SearchData());
}
```

The *initSearchData* method takes *sd*, the SearchData instance to initialize to the root, and returns the initialized SearchData instance. Separating this from the *createSearchData* method allows the initialization process to be independently overridden. This method is called by the *allocate* method in the search data pool so that the search data object returned is properly initialized even if it has been used previously.

```
SearchData initSearchData(SearchData sd){
    sd.ptr = root;
    sd.bp = 0;
    return sd;
}
```

Correctness Highlights: By definition the branch position for the root is 0. ■

The *newLeafNode* method takes *element*, the element to place in a new leaf node. It returns a reference to a new leaf node holding the given element.

```
TrieLeafNode<E> newLeafNode(E element){
    return new LeafNode(element);
}
```

41.7.2 Algorithmic Accessors

The *find* method takes *element*, the target and *sd*, a SearchData instance to hold the ending location for the search. It returns the find result from the search.

```
FindResult find(E element, SearchData sd){
    if (isEmpty())                                     //for efficiency
        return FindResult.UNMATCHED;
    initSearchData(sd);                               //start sd at the root
    int numDigitsInElement = digitizer.numDigits(element);
    while (sd.ptr != null && !sd.atLeaf()) {           //while a child exists
        if (sd.bp == numDigitsInElement)               //Case 1
            return FindResult.PREFIX;
        if (sd.moveDown(element) == NO_CHILD)         //Case 2
            return FindResult.UNMATCHED;
    }
    if (sd.ptr != null && sd.bp != numDigitsInElement) //Case 3a
        return FindResult.EXTENSION;
    else                                           //Case 3b
        return FindResult.MATCHED;
}
```

Correctness Highlights: By PATH and REACHABLE, if *element* is in the collection then *find* will end at a leaf containing data equivalent to *element*, so *MATCHED* is returned. By BRANCH-POSITION and the correctness of *moveDown*, *sd* is at *x* if and only if the digits processed in *element* are equal to *sp(x)*. We now discuss the cases that can occur:

Case 1: Search ends at internal node x . This case occurs only if all digits of *element* (not counting the end of string character) have been processed, since otherwise either a *null* child or a leaf would be reached. For example, consider a search for da# in the trie of Figure 41.1. By PATH when this case occurs, *element* is a prefix of all descendants of x . Thus *PREFIX* is the correct return value.

Case 2: Search ends at a *null* child. First, this case occurs exactly when the search data *move-Down* method returns *NO_CHILD*. By PATH when this case occurs, *element* is not in the collection, so *UNMATCHED* is a correct return value. As an example, of this case, consider a search for cafe# (shown as a bold search path) in the trie of Figure 41.1.

Case 3: Search ends at leaf ℓ . We further divide this into the following three subcases.

Case 3a: ℓ is reached with at least one digit of *element* not yet processed. By PATH, this implies that *element* is an extension of the element held in ℓ , so *EXTENSION* is the correct return value.

Case 3b: ℓ is reached with all digits of *element* processed. By PATH, this occurs only when the element is in the collection. So *MATCHED* is the correct return value. ■

The method *contains* takes *element*, the element being tested for membership in the collection, and returns *true* if and only if an equivalent element exists in the collection.

```
public boolean contains(E element) {
    if (isEmpty()) //for efficiency
        return false;
    SearchData sd = pool.allocate(); //allocate sd from pool
    try {
        return find(element, sd) == FindResult.MATCHED;
    } finally {
        pool.release(sd); //release sd to pool
    }
}
```

The method *get* takes r , the desired rank. It returns the r^{th} element in the sorted order, where $r = 0$ is the minimum. It throws an *IllegalArgumentException* when $r < 0$ or $r \geq n$.

```
public E get(int r) {
    if (r < 0 || r ≥ getSize())
        throw new IllegalArgumentException();
    Locator<E> loc = iterator();
    for (int j=0; j < r+1; j++, loc.advance());
    return loc.get();
}
```

Correctness Highlights: By the correctness of the locator methods, and the fact that the iteration order is based on the sorted order, advancing $r + 1$ times from FORE leaves the locator at the rank r element. ■

The *getEquivalentElement* method takes *element*, the target, and returns an equivalent element in the collection. It throws a *NoSuchElementException* when there is no equivalent element in the collection.

```

public E getEquivalentElement(E element) {
    SearchData sd = pool.allocate();
    try {
        if (find(element, sd) == FindResult.MATCHED)
            return sd.ptr.data();
        else
            throw new NoSuchElementException();
    } finally {
        pool.release(sd);
    }
}

```

The *min* method returns the least element in the collection, as defined by the digitizer. It throws a *NoSuchElementException* when the collection is empty.

```

public E min() {
    if (isEmpty())
        throw new NoSuchElementException();
    else
        return FORE.next().data();
}

```

Correctness Highlights: Follows from ORDEREDLEAFCHAIN, and the correctness of the *isEmpty*, *next*, and *data* methods. ■

The *max* method returns the greatest element in the collection, as defined by the digitizer. It throws a *NoSuchElementException* when the collection is empty.

```

public E max() {
    if (isEmpty())
        throw new NoSuchElementException();
    else
        return AFT.prev().data();
}

```

Correctness Highlights: Follows from ORDEREDLEAFCHAIN and the correctness of the *isEmpty*, *prev*, and *data* methods. ■

The **predecessor** of *e*, is defined as the greatest element in the collection that is less than *e*. On completion of a successful or unsuccessful search, it is often necessary to move to the node that is the predecessor of the ending search location. For example, to insert a new element, *e*, one must find the predecessor of *e*. Similarly, the public successor and predecessor methods sometimes need to find the predecessor of an element not in the collection. To support these methods, the internal *moveToPred* method takes *element*, the target, *sd*, a SearchData instance that has already been set by *find(element)*, and *findStatus*, the return value from *find(element)*. If there is some element in the collection less than *element* then *sd* is moved to the predecessor. Otherwise, *sd* is not changed. This method returns *true* if there is some element in the collection less than *element* and otherwise returns *false*.

```

protected boolean moveToPred(E element, SearchData sd, FindResult findStatus){
    if (sd.atLeaf() && (findStatus == FindResult.GREATER ||
        findStatus == FindResult.EXTENSION))
        return true;                                     //Case 1
    if (findStatus != FindResult.GREATER)                 //Case 2 and Case 3
        sd.retraceToLastLeftFork(element);
    if (sd.atRoot())                                       //Rest of Cases 2, 3 and Case 4
        return false;
    else if (!sd.atLeaf())
        sd.moveToMaxDescendant();
    return true;
}

```

Correctness Highlights: One of the following three cases must occur.

Case 1: *sd* is at a leaf for which *element* is lexicographically greater than the element at the search data location. This case occurs exactly when the *findStatus* is *GREATER* or *EXTENSION*. By *ORDEREDLEAFCHAIN* it follows that *sd.ptr* references the predecessor. Thus, *sd* need not be moved, and *true* should be returned.

Case 2: The search for *element* ended at a *null* child, leaving *sd* at the node from which the *null* child was found. For example, consider finding the predecessor for *deaf#* in the trie shown in Figure 41.1. The search path for *find* is shown as a dashed line in Figure 41.1. In this case, *retraceToLastLeftFork* is used to find the lowest node on the search path for which there is a non-*null* left child. We further divide Case 2 into the following two subcases.

Case 2a: After *retraceToLastLeftFork*, *sd* is at the root. For example, this would occur when finding the predecessor of *ad#* in the Trie of Figure 41.1 By the correctness of *retraceToLastLeftFork*, in this case there is no element in the collection smaller than *element*. Thus *false* is the correct return value.

Case 2b: Otherwise (as in finding the predecessor of *deaf#*), by the correctness of *retraceToLastLeftFork* it will move *sd* to the left fork root. By the correctness of *moveToMaxDescendant*, upon completion *sd* will be at the predecessor of *element* and *true* is correctly returned.

Case 3: The search for *element* ended at a node *x* that is either a leaf node holding *element* or a node for which *element* is lexicographically smaller than all elements in $T(x)$. As in Case 2, the predecessor will be the largest element in the rightmost non-empty subtree that is left of the *null* child where the search ended.

Case 4: *sd* is at a node *x* for which *element* is lexicographically greater than all elements in $T(x)$. For example, consider when *element* is *deaf#* in the compressed trie of Figure 39.4. By *PATH* it follows that *element* would be in $T(x)$. Together with the fact that *element* is greater than all elements in $T(x)$ implies that the predecessor of *element* is the largest element in $T(x)$. Finally, *sd* would only be at the root when *element* is the empty string in which case there is no predecessor. The rest of the correctness is like that of Case 2b.

The rest of the correctness for this case follows from that of the search data *atLeaf*, *retraceToLastLeftFork*, and *moveToMaxDescendant* methods. ■

The public *predecessor* method takes *element*, the element for which to find the predecessor. It returns the maximum element in the ordered collection that is less than *x*. This method does not require that *element* be in the collection. It throws a *NoSuchElementException* when no element in the collection is less than *element*.

```
public E predecessor(E element) {
    if (isEmpty())
        throw new NoSuchElementException();
    SearchData sd = pool.allocate();
    try {
        if (moveToPred(element, sd, find(element, sd)))
            return sd.ptr.data();
        else
            throw new NoSuchElementException();
    } finally {
        pool.release(sd);
    }
}
```

The public *successor* method takes *element*, the element for which to find the successor. It returns the least element in the collection that is greater than *element*. This method does not require that *element* be in the collection. It throws a *NoSuchElementException* when no element in the collection is greater than *element*.

```
public E successor(E element) {
    if (isEmpty()) //for efficiency
        throw new NoSuchElementException();
    SearchData sd = pool.allocate();
    try {
        FindResult findStatus = find(element, sd);
        TrieLeafNode<E> succ = AFT; //init. successor to AFT
        if (findStatus == FindResult.MATCHED) //Case 1
            succ = ((TrieLeafNode<E>) sd.ptr).next();
        else if (moveToPred(element, sd, findStatus)) //Case 2
            succ = ((TrieLeafNode<E>) sd.ptr).next();
        if (succ == AFT)
            throw new NoSuchElementException();
        else
            return succ.data();
    } finally {
        pool.release(sd);
    }
}
```

Correctness Highlights: We consider the following cases.

Case 1: The given element is in the collection. By the correctness of *find*, *findStatus* will be *MATCHED* and the search location will be at the leaf node holding the given element. By *ORDEREDLEAFCHAIN*, the element that follows *sd.ptr* in the ordered leaf chain is the correct return value.

Case 2: We know that *moveToPred* returns *true* if and only if *element* is not the minimum ele-

ment in the collection. Also, when *true* is returned, *moveToPred* moves the search location to the predecessor of *element*. As in Case 1, the rest follows from ORDEREDLEAFCHAIN.

Case 3: If neither of the above two cases occur, then *element* is smaller than the minimum element in the collection, so leaving *succ* as AFT is correct. ■

The internal *moveToLowestMatchingAncestor* method takes *prefix*, the desired prefix, *sd*, a SearchData instance that has been set by *find(prefix)*, and *findStatus*, the FindResult value returned by the search used to set *sd*. This method has the side affect of moving *sd* to its lowest ancestor for which the associated data is an extension of *prefix*. Since a trie has an internal node that corresponds to each digit of *prefix*, this method is very straightforward and does not use *findStatus*. However, the subclasses of Trie will override it.

The only way in which the current location for *sd* is not already at the lowest common ancestor is when the end of string character from a PrefixFreeDigitizer has been processed. For example, consider the case where *prefix* is fee# for the Trie in Figure 41.1. In such a case, the lowest common ancestor is the parent of the current position.

```
protected void moveToLowestCommonAncestor(E prefix, SearchData sd,
                                           FindResult findStatus) {
    if (sd.processedEndOfString(prefix))
        sd.moveUp();
}
```

Correctness Highlights: Follows from PATH and the correctness of the SearchData *processedEndOfString* and *moveUp* methods. ■

The *completions* method takes *prefix*, the desired element for which all completions are sought, and *c*, a collection to which all elements in the collection that consists of *prefix* followed by any number of (possibly zero) additional digits are added.

```
public void completions(E prefix, Collection<? super E> c) {
    if (isEmpty())                                     //for efficiency
        return;
    SearchData sd = pool.allocate();
    try {
        FindResult findStatus = find(prefix, sd);
        int numDigits = digitizer.numDigits(prefix);
        if (digitizer.isPrefixFree()) {
            numDigits--;
            if (sd.processedEndOfString(prefix))        //don't process end of string
                sd.moveUp();
        }
        if (findStatus == FindResult.PREFIX || findStatus == FindResult.MATCHED ||
            sd.numMatches() == numDigits)
            sd.elementsInSubtree(c);
        else
            return;
    } finally {
        pool.release(sd);
    }
}
```

Correctness Highlights: Clearly if the collection is empty then *null* is the correct return value. We now consider when the collection is not empty. By the correctness of the *find* method, *sd* and *findStatus* are correctly set. After the first conditional in the try block has been executed, *numDigits* is set to the number of digits in *prefix* excluding the end of string character. Also, *sd* holds the location where the search ends when processing all of *prefix* except for the end of string character.

It is easily seen that there is a completion of *prefix* exactly when *found* is *PREFIX* or *MATCHED*, or when all digits of *prefix* (excluding the end of string character) are processed. Finally, when there is a completion, by *PATH* all completions in the collection are in the subtree rooted at *sd*. The rest follows from the correctness of the *SearchData elementsInSubtree* method. ■

The *longestCommonPrefix* method takes *prefix*, the desired prefix, and *c*, a collection to which all elements in the collection that have a longest common prefix with *element* are added.

```
public void longestCommonPrefix(E prefix, Collection<? super E> c) {
    if (isEmpty())                                     //for efficiency
        return;
    SearchData sd = pool.allocate();
    try {
        moveToLowestCommonAncestor(prefix, sd, find(prefix, sd));
        sd.elementsInSubtree(c);
    } finally {
        pool.release(sd);
    }
}
```

Correctness Highlights: Clearly if the collection is empty then *null* is the correct return value. By *PATH*, when the collection is not empty, all elements in the subtree rooted at the lowest common ancestor will be longest prefix matches with the given prefix. The rest of the correctness of *moveToLowestMatchingAncestor* and the *SearchData elementsInSubtree* method. ■

41.7.3 Content Mutators

Methods to Perform Insertion

To insert *element*, first an unsuccessful search for *element* is performed. Then an internal node must be added for each character of *element* that was not processed. Also, the ordered leaf chain must be maintained. For example, [Figure 41.3](#) shows the trie that results when inserting *cafe#* into the trie of [Figure 41.1](#). Also *cafe#* is inserted in the ordered leaf chain between *cab#* and *dab#*.

The *addNewNode* method takes *newNode*, a trie node to add that already holds the new element, and *sd*, a *SearchData* instance set by a call to *find(newNode.data())*. It modifies the trie (excluding the ordered leaf chain) to include *newNode*. It requires that adding the new element will preserve the property that the collection is prefix free.

```
protected void addNewNode(TrieNode<E> newNode, SearchData sd){
    E element = ((LeafNode) newNode).data();
```

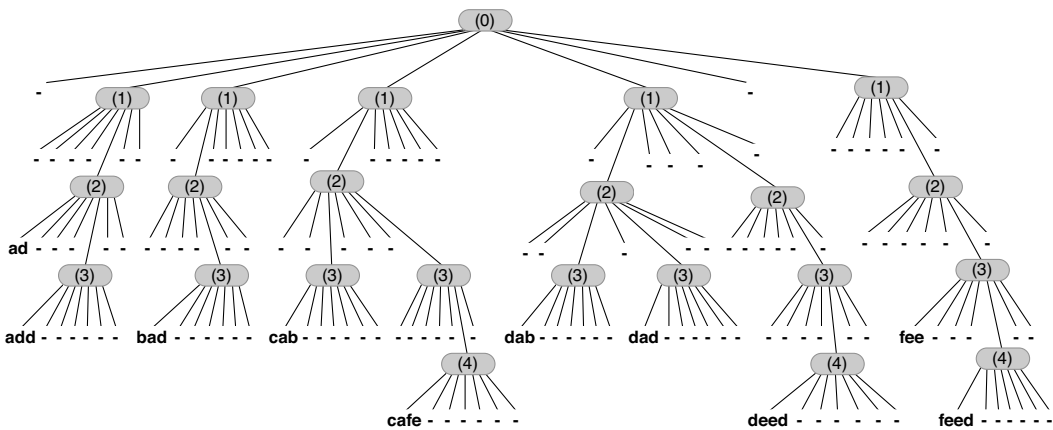



Figure 41.3

The Trie that results when inserting `cafe#` into the Trie shown in Figure 41.1.

```

if (sd.ptr == null) {                                     //collection is currently empty
    root = new InternalNode();
    sd.ptr = root;
}
while (sd.bp < digitizer.numDigits(element) - 1) {
    InternalNode internalNode = new InternalNode();
    ((InternalNode) sd.ptr).setChild(internalNode, element, sd.bp++);
    sd.ptr = internalNode;
}
((InternalNode) sd.ptr).setChild(newNode, element, sd.bp++);
sd.ptr = newNode;
}

```

Correctness Highlights: By the correctness of the *find* method and the requirement placed on *sd*, it follows that *sd* is at the last non-null element in the search path defined *element*. If *sd.ptr* is *null*, it follows that the collection is currently empty in which case the root is set to the new node and *sd* is modified to be at the root.

To maintain `BRANCHPOSITION` it is necessary to have an internal node for each digit in *element* except for the last one (that leads to the leaf *newNode*). The `TrieNode setChild` method maintains `PATH`, `PARENT`, and `NUMCHILDREN`. ■

The internal *insert* method takes *element*, the element to be added to the collection. It returns a reference to the newly inserted leaf node. It throws an *IllegalArgumentException* when adding *element* to the collection would violate the requirement that no element in the collection is a prefix of any other element in the collection. Recall that two equal elements are considered to be prefixes of each other.

```

protected TrieLeafNode<E> insert(E element) {
    SearchData sd = pool.allocate();
    try {
        FindResult found = find(element, sd);
        if (found == FindResult.MATCHED || (!digitizer.isPrefixFree() &&
            (found == FindResult.PREFIX || found == FindResult.EXTENSION)))

```

```

        throw new IllegalArgumentException(element + " violates prefix-free requirement");
        TrieLeafNode<E> newNode = newLeafNode(element);    //create the new leaf node
        addNewNode(newNode, sd);                          //add it to the trie (sd is updated)
        found = FindResult.MATCHED;                      //update found (to avoid a new search)
        if (moveToPred(element, sd, found))               //maintain OrderedLeafChain Property
            newNode.addAfter((TrieLeafNode<E>) sd.ptr);
        else
            newNode.addAfter(FORE);
        size++;                                          //preserve Size Property
        return newNode;                                //return reference to the new leaf node
    } finally {
        pool.release(sd);
    }
}

```

Correctness Highlights: By the correctness of *find*, if *found* is either *PREFIX*, *EXTENSION*, or *MATCHED*, then inserting *element* would cause an element in the collection to be a prefix of any other element. So an exception is properly thrown in these cases.

In the remaining cases, a new leaf node is created and added to the trie. By the correctness of *addChild* (called by *addNewNode*), *PATH*, *PARENT*, and *NUMCHILDREN* are maintained. As discussed in the correctness highlights for *addNewNode*, that method preserves *BRANCH-POSITION*. By the correctness of *find*, *sd* references a node reachable from the root. Since *addNewNode* inserts the new node so that it is reachable from *sd*, *REACHABLE* is maintained.

Recall that *addNewNode* modifies *sd* to reference the new node. Also, *found* is correctly updated to *MATCHED* since *sd* references a trie node holding *element*. These properties are important, since *moveToPred* requires that *sd* and *found* are set by *find*. Although *find* is not called after the trie is modified, we have ensured that both *sd* and *found* contain the values that would have been obtained by calling *find*, which avoids unnecessary computation.

By the correctness of *moveToPred*, if *true* is returned, then a predecessor existed and *sd* is now referencing the predecessor. In this case, the new element should be linked immediately after its predecessor in the leaf chain. Similarly, if *moveToPred* returns *false* then *element* is the first element in the iteration order, so it should be inserted after *FORE* in the leaf chain. Thus in both cases, by the correctness of *addAfter*, *ORDEREDLEAFCHAIN* is preserved. Incrementing *size* preserves *SIZE*.

REDIRECTCHAIN is preserved since the nodes not in use are unchanged. Finally, since *addAfter* sets *newNode.prev* to the predecessor, *INUSE* is maintained. ■

The public *add* method takes *element*, the new element, and inserts *element* into the collection.

```

public void add(E element) {
    insert(element);
}

```

The public *addTracked* method takes *element*, the new element, and inserts *element* into the collection. It returns a locator that tracks the newly added element.

```

public Locator<E> addTracked(E element) {
    return new Tracker(insert(element));
}

```

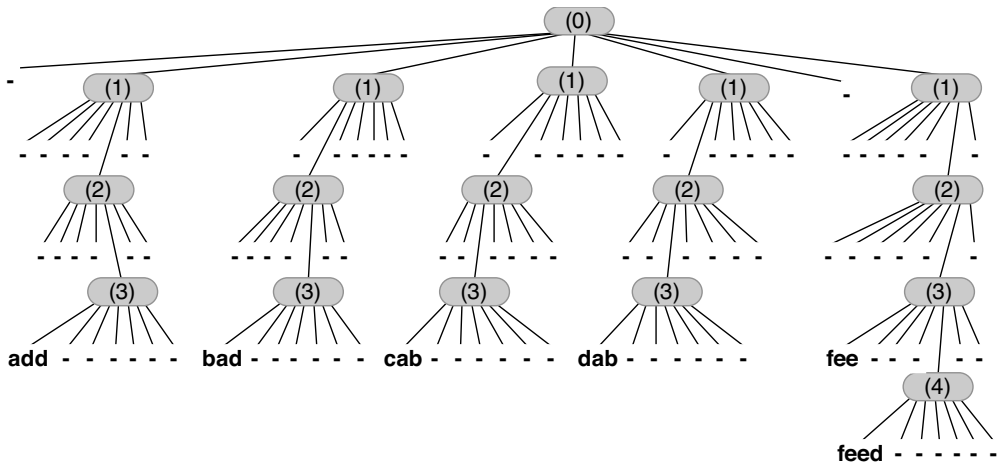


Figure 41.4

The Trie that results when removing `ad#`, `deed#`, and `dad#` from the Trie shown in Figure 41.1.

Methods to Perform Deletion

The first step in removing an element from a trie is to find the corresponding leaf node ℓ . Then the reference to ℓ is replaced by *null*. If that leaves an internal node with no children, then that node is replaced by a *null*. This process is repeated as long as it leaves an internal node with no children.

As an example, we consider removing `ad#`, `deed#`, and `dad#` from the trie in Figure 41.1. Removing `ad#` is performed by just replacing the reference to the leaf node with *null*. Removing `deed#` causes three internal nodes to be removed. Finally, to remove `dad#` one internal node is also replaced by *null*. The resulting trie is shown in Figure 41.4. The internal *remove* method takes *node*, a reference to the trie node to remove.

```
void remove(TrieNode<E> node){
    ((LeafNode) node).remove();           //preserve OrderedLeafChain
    E element = ((LeafNode) node).data();  //element in node to remove
    int level = digitizer.numDigits(element); //current level in trie
    do {
        InternalNode parent = (InternalNode) node.parent(); //parent of current value for node
        level--;                                           //adjust to parent's level
        parent.children[parent.childIndex(element, level)] = null; //remove node from trie
        parent.numChildren--;                             //preserve NumChildren for parent
        node = parent;                                    //move to parent
    } while (node != root && ((InternalNode) node).numChildren == 0);
    size--;                                              //preserve Size property
}
```

Correctness Highlights: By ORDEREDLEAFCHAIN and the correctness of the leaf node *remove* method, INUSE, ORDEREDLEAFCHAIN, and REDIRECTCHAIN are preserved. It is easily verified that the body of the do loop removes *node* from the trie. The initial execution of the loop body preserves REACHABLE. Repeating this process until reaching either the root or a node with at least one child preserves BRANCHPOSITION.

SIZE is preserved by decrementing of *size*. The remaining properties are not affected since

the structure of the trie is unchanged for all but the removed node (and possibly its ancestors as discussed above). ■

The *removeImpl* method takes *sd*, a search data instance that has been set by a search for the element to be removed. It calls an internal method that removes *sd.ptr* from the trie.. This method requires that the structure of the Patricia trie has not changed since *find* was called with *sd.ptr.data*. We separate this simple method from the public *remove* method so that the PatriciaTrie class (Chapter 44) can override it.

```
protected void removeImpl(SearchData sd) {
    remove(sd.ptr);
}
```

The public *remove* method takes *element*, the element to remove. It removes from the collection the element (if any) equivalent to *element*. It returns *true* if an element was removed, and *false* otherwise.

```
public boolean remove(E element) {
    if (isEmpty())                                //for efficiency
        return false;
    SearchData sd = pool.allocate();
    try {
        if (find(element, sd) != FindResult.MATCHED)
            return false;
        removeImpl(sd);
        return true;
    } finally {
        pool.release(sd);
    }
}
```

41.7.4 Locator Initializers

The *iterator* method creates a new tracker that is at FORE.

```
public Locator<E> iterator() {
    return new Tracker(FORE);
}
```

Correctness Highlights: The correctness of the iterator follows from the correctness of the locator methods and the fact that *FORE* is the head of the ordered leaf chain. ■

The *iteratorAtEnd* method creates a new tracker that is at AFT.

```
public Locator<E> iteratorAtEnd() {
    return new Tracker(AFT);
}
```

Correctness Highlights: The correctness of the iterator follows from the correctness of the locator methods and the fact that *AFT* is the tail of the ordered leaf chain. ■

The method *getLocator* takes *x*, the element to track. It returns a new tracker that is initialized to track *x*. It throws a *NoSuchElementException* when *x* is not in the collection.

```

public Locator<E> getLocator(E x) {
    SearchData sd = pool.allocate();
    if (find(x, sd) == FindResult.MATCHED) {
        TrieLeafNode<E> t = (TrieLeafNode<E>) sd.ptr;
        pool.release(sd);
        return new Tracker(t);
    }
    else
        throw new NoSuchElementException();
}

```

41.8 Trie Tracker Inner Class

AbstractCollection<E>.AbstractLocator<E> implements *Locator*<E>

└ **Tracker** implements *Locator*

Each trie tracker has an instance variables *node* that references the trie node holding the tracked element.

```
TrieLeafNode<E> node;
```

The constructor takes a single argument, *ptr*, a pointer to the node to track.

```

Tracker(TrieLeafNode<E> ptr) {
    this.node = ptr;
}

```

Recall that *inCollection* returns *true* if and only if the tracked element is currently in the collection.

```

public boolean inCollection() {
    if (node == FORE || node == AFT)
        return false;
    return !node.isDeleted();
}

```

Correctness Highlights: Follows from INUSE and the correctness of the *TrieNode isDeleted* method. ■

The *get* method returns the tracked element. It throws a *NoSuchElementException* when this tracker is not at an element in the collection.

```

public E get() {
    if (!inCollection())
        throw new NoSuchElementException();
    return (E) node.data();
}

```

The internal *skipRemovedElements* method takes *ptr*, reference to a trie node that is no longer in the collection, and returns a reference to the element in the collection that follows the element referenced by *ptr* in the iteration order. If there is no such element AFT is returned. Similar to the path compression performed by the union-find data structure (Section 6.3), this method performs the optimization of compressing the path of the redirect chain by updating all next pointers to refer directly to the returned element.

```
protected TrieLeafNode<E> skipRemovedElements(TrieLeafNode<E> ptr) {
    if (ptr == FORE || ptr == AFT || !ptr.isDeleted())
        return ptr;
    ptr.setNext(skipRemovedElements(ptr.next()));
    return ptr.next();
}
```

Correctness Highlights: If *ptr* references an element in the collection, FORE, or AFT, *ptr* is the correct return value. Otherwise, once an element in the collection is reached by following the next pointers, by REDIRECTCHAIN, we have reached the element in the collection that follows the tracker in the iteration order. Setting *ptr.next* to the value returned by the recursive call performs the path compression. Termination is guaranteed by REDIRECTCHAIN. ■

The *advance* method moves this tracker to the next element in the iteration order (or AFT if the tracker is currently at the last element). It returns *true* if and only if, after the update, the tracker is at an element of the collection. It throws an *AtBoundaryException* when this tracker is already at AFT since there is no place to advance.

```
public boolean advance() {
    if (node == AFT) //Case 1
        throw new AtBoundaryException();
    if (node != FORE && node.isDeleted()) //Case 2
        node = skipRemovedElements(node);
    else //Case 3
        node = node.next();
    return node != AFT;
}
```

Correctness Highlights: We consider the following three cases:

Case 1: The tracker is at AFT. By the specification, an *atBoundaryException* is thrown in this case.

Case 2: The tracker is at a node holding an element that has been removed from the collection. Observe that this case occurs exactly when the tracked node is neither FORE nor AFT, and it is deleted. By the correctness of *skipRemovedElements*, *node* will be left at the element in the collection that follows the tracked node.

Case 3: The tracker is at a node holding an element in the collection. By the correctness of *succ*, the tracker is moved to the next element in the iteration order. ■

The *retreat* method moves this tracker to the previous element in the iteration order (or FORE if the tracker is currently at the first element). It returns *true* if and only if, after the update, the tracker is at an element of the collection. It throws an *AtBoundaryException* when this tracker is already at FORE since there is no place to retreat.

```

public boolean retreat() {
    if (node == FORE)
        throw new AtBoundaryException();
    if (node != AFT && node.isDeleted())
        node = skipRemovedElements(node);
    node = node.prev();
    return node != FORE;
}

```

Correctness Highlights: Like that for *advance*, except that when the tracked node has been deleted, *skipRemovedElements* returns the successor. Thus we move to the previous element except when the tracker is at FORE. ■

The *hasNext* method that returns *true* if there is some element in the collection after the currently tracked element.

```

public boolean hasNext() {
    if (node.isDeleted())
        skipRemovedElements(node);
    return (node != AFT && node.next() != AFT);
}

```

Correctness Highlights: Like that for *advance*, except it is simpler since the value of *node* is not updated. ■

As discussed in Section 5.8, the *remove* method removes the tracked element, leaving the tracker logically between the elements in the iteration order that preceded and followed the one removed. It throws a *NoSuchElementException* when the tracker is at FORE or AFT.

```

public void remove() {
    if (!inCollection())
        throw new NoSuchElementException();
    Trie.this.remove(node);
}

```

Correctness Highlights: Follows from the *TrieLeafNode remove* method that preserves INUSE and REDIRECTCHAIN. ■

41.9 Performance Analysis

The asymptotic time complexities of all public methods for the *Trie* class are shown in Table 41.5, and the asymptotic time complexities for all of the public methods of the *Trie Tracker* class are given in Table 41.6.

For a trie node x , we let h_x be defined as the length of the path from the root to x . The **height** of a trie is defined as the maximum over all nodes x in the trie of h_x . Throughout this section, for a digitized element o we use d_o to denote the number of digits in o . When it is clear from context, we just use d . Let S be the set of elements in the collection. We define $d_{\max} = \max_{o \in S} d_o$. A fundamental property of the trie that follows from LEAFPATH is that the height of the trie is bounded above by $d_{\max}$ since the length of the path to any element is the number of digits in that element.

Thus, the worst-case height for the trie is $O(d_{\max})$. Knuth [97] gives a detailed analysis proving that for a trie built from n random elements (where the number of digits approaches infinity), the expected number of digit comparisons to distinguish a random element from those in the collection is $O(\log_b n) = O(\log n / \log b)$. Thus, for the trie holding n random elements, the expected value for $h = O(\log_b n) = O(\log n / \log b)$.

To approximate the space complexity of a trie, we build upon the result of Knuth [97], who showed that for a compact trie holding random elements the expected number of internal nodes is approximately $n / \ln b$. As a very easy lower bound, observe that a trie has, for each element, at least one internal node that is not in the corresponding compact trie. Thus the expected number of internal nodes for a trie is at least $n + n / \ln b = n(1 + 1 / \ln b)$.

Since a trie is an elastic implementation, *ensureCapacity* and *trimToSize* need not perform any computation, so they take constant time. It is also easily seen that the constructor, *iterator*, and *iteratorAtEnd* take constant time.

The *min* method takes a single pass down the tree, always going to the left child, and the *max* method takes a single pass down the tree always going to the right child. Thus both take $O(h)$ time. The rest follows from applying the derived bound for h .

The cost of a successful search for o is $O(d_o)$ regardless of the structure of the trie. The time complexity for *contains(o)*, *getEquivalentElement(o)*, *getLocator(o)* is the cost of the successful search. The time required by *remove* after locating the desired element is bounded by the number of digits in the element being removed since it just involves, at worst, a pass back towards the root.

An unsuccessful search for o has worst-case cost $O(\min(d_o, d_{\max}))$ since it completed when either all digits of o have been processed, or when reaching a leaf. The expected cost for an unsuccessful search is $O(h) = O(\log n / \log b)$. The *moveToPred* method takes $O(h)$ time since in the worst case it must move up to the root and then down to the predecessor. Thus, *predecessor* and *successor* take $O(h)$ time. The *add* and *addTracked* methods begin with an unsuccessful search. The cost of adding the new element to the trie is bounded by the number of digits in the element being added, since that bounds the number of new elements. Also, both *add* and *addTracked* must find the predecessor* to maintain the ordered leaf chain. Thus, these methods also take $O(h)$ time.

The *accept*, *clear*, *toString*, and *toArray* methods perform an inorder traversal. Observe that during the traversal each node is visited exactly once and constant time is spent at each node. Thus the overall cost is $O(n)$.

For our analysis of *addAll* and *retainAll*, we assume that $d_{\max}$ is an upper bound on the number of digits in any element from the collection on which it is called, and on the number of digits in any element from the collection c . The *addAll* method calls *add* for each element in the provided collection. Similarly, *retainAll* performs a search in c for each element in the trie (which we assume takes $O(n \cdot |c|)$ time, and then possibly removes each element from the collection on which it is called. The stated bounds both follow by just applying the bounds for adding or removing a single element. for each element in collection c .

We now analyze the time complexity of the locator methods. By using the ordered leaf chain, navigation forwards or backwards in the iteration order can be performed in constant time. The only method that requires additional computation is *remove*. Although no search is required to locate the element to remove, the branch position of the element cannot be saved as part of the *TrieNode* since it can change as a side effect of other mutations. Thus, a search must be performed in order to compute the branch position. The remaining steps to remove an element take constant time. Thus, the worst-case cost for *remove* is $O(d)$. As with the *Trie remove* method, one can also show that the locator *remove* method has expected cost $O(\min(d, \log n / \log b))$ for a randomly constructed trie.

*As discussed under the optimizations, one could reduce the time complexity of *add* and *addTracked* to $O(d)$ by not maintaining the ordered leaf chain. The cost of such a change would be to increase the time complexity for the locator methods that navigate through the ordered collection.

method	worst-case time complexity	expected time complexity
constructor	$O(1)$	$O(1)$
<i>ensureCapacity</i> (<i>x</i>)	$O(1)$	$O(1)$
<i>iterator</i> ()	$O(1)$	$O(1)$
<i>iteratorAtEnd</i> ()	$O(1)$	$O(1)$
<i>trimToSize</i> ()	$O(1)$	$O(1)$
<i>add</i> (<i>o'</i>)	$O(\min(d, d_{\max}))$	$O(\log n / \log b)$
<i>addTracked</i> (<i>o'</i>)	$O(\min(d, d_{\max}))$	$O(\log n / \log b)$
<i>contains</i> (<i>o'</i>), unsuccessful	$O(\min(d, d_{\max}))$	$O(\log n / \log b)$
<i>min</i> ()	$O(d_{\max})$	$O(\log n / \log b)$
<i>max</i> ()	$O(d_{\max})$	$O(\log n / \log b)$
<i>predecessor</i> (<i>o'</i>)	$O(d_{\max})$	$O(\log n / \log b)$
<i>successor</i> (<i>o'</i>)	$O(d_{\max})$	$O(\log n / \log b)$
<i>contains</i> (<i>o</i>), successful	$O(d)$	$O(d)$
<i>getEquivalentElement</i> (<i>o</i>)	$O(d)$	$O(d)$
<i>getLocator</i> (<i>o</i>)	$O(d)$	$O(d)$
<i>remove</i> (<i>o</i>)	$O(d)$	$O(d)$
<i>completions</i> (<i>o'</i>)	$O(d_{\max} + s)$	$O(\log n / \log b + s)$
<i>longestCommonPrefix</i> (<i>o'</i>)	$O(d_{\max} + s)$	$O(\log n / \log b + s)$
<i>clear</i> ()	$O(n)$	$O(n)$
<i>toArray</i> ()	$O(n)$	$O(n)$
<i>toString</i> ()	$O(n)$	$O(n)$
<i>accept</i> ()	$O(n)$	$O(n)$
<i>addAll</i> (<i>c</i>)	$O(c \cdot d_{\max})$	$O(c \cdot \log_b(n + c))$
<i>retainAll</i> (<i>c</i>)	$O(n(c + d_{\max}))$	$O(n(c + \log n / \log b))$

Table 41.5 Summary of the asymptotic time complexities for the DigitizedOrderedCollection public methods when using the trie data structure. We use the convention that *o* is in the collection and *o'* is not in the collection. The expected time complexities are under the assumption that all elements in the trie and *o*, and *o'*, are random elements where the number of digits approaches infinity. This column shows the expected behavior as a function of *n* when the number of digits is typically larger than $\log_b n$. We use *d* to denote the number of digits in the object *o* (or *o'*), *d*_{max} to denote the maximum number of digits in any element in the collection, and *s* to denote the number of elements in a positional collection returned.

locator method	time complexity
constructor	$O(1)$
<i>advance</i> ()	$O(1)$
<i>get</i> ()	$O(1)$
<i>hasNext</i> ()	$O(1)$
<i>next</i> ()	$O(1)$
<i>retreat</i> ()	$O(1)$
<i>remove</i> ()	$O(d)$

Table 41.6 Summary of the time complexities for the public Locator methods of the Trie Locator class. We use *d* to denote the number of digits in the element to be removed.

41.10 Quick Method Reference

Trie Public Methods

- p. 650** *Trie*(Digitizer<? super E> digitizer)
- p. 98 void **accept**(Visitor<? super E> v)
- p. 659** void **add**(E element)
- p. 100 void **addAll**(Collection<? extends E> c)
- p. 659** Locator<E> **addTracked**(E element)
- p. 100 void **clear**()
- p. 656** void **completions**(E prefix, Collection<? super E> c)
- p. 97 boolean **contains**(E value)
- p. 99 void **ensureCapacity**(int capacity)
- p. 652** E **get**(int r)
- p. 96 int **getCapacity**()
- p. 96 Comparator<? super E> **getComparator**()
- p. 98 E **getEquivalentElement** (E target)
- p. 662** Locator<E> **getLocator**(E x)
- p. 96 int **getSize**()
- p. 96 boolean **isEmpty**()
- p. 661** Locator<E> **iterator**()
- p. 661** Locator<E> **iteratorAtEnd**()
- p. 657** void **longestCommonPrefix**(E prefix, Collection<? super E> c)
- p. 653** E **max**()
- p. 653** E **min**()
- p. 655** E **predecessor**(E element)
- p. 661** boolean **remove**(E element)
- p. 100 void **retainAll**(Collection<E> c)
- p. 655** E **successor**(E element)
- p. 97 Object[] **toArray**()
- p. 97 E[] **toArray**(E[] array)
- p. 98 String **toString**()
- p. 99 void **trimToSize**()

Trie Internal Methods

- p. 657** void **addNewNode**(TrieNode<E> newNode, SearchData sd)
- p. 97 int **compare**(E e1, E e2)
- p. 650** SearchData **createSearchData**()
- p. 97 boolean **equivalent**(E e1, E e2)
- p. 651** FindResult **find**(E element, SearchData sd)
- p. 651** SearchData **initSearchData**(SearchData sd)
- p. 658** TrieLeafNode<E> **insert**(E element)
- p. 656** void **moveToLowestCommonAncestor**(E prefix, SearchData sd, FindResult findStatus)
- p. 653** boolean **moveToPred**(E element, SearchData sd, FindResult findStatus)
- p. 651** TrieLeafNode<E> **newLeafNode**(E element)
- p. 660** void **remove**(TrieNode<E> node)
- p. 661** void **removeImpl**(SearchData sd)
- p. 99 void **traverseForVisitor**(Visitor<? super E> v)
- p. 98 void **writeElements**(StringBuilder s)

Trie.InternalNode Public Methods

- p. 636 TrieNode<E> **child**(int i)
- p. 643** int **childIndex**(E element, int bp)
- p. 636 E **data**()

- p. 636 boolean **isLeaf**()
- p. 636 TrieNode<E> **parent**()
- p. 636 void **setParent**(TrieNode<E> parent)

Trie.InternalNode Internal Methods

- p. 643** *InternalNode*()
- p. 643** int **setChild**(TrieNode<E> child, E element, int bp)

Trie.LeafNode Public Methods

- p. 638 void **addAfter**(TrieLeafNode<E> ptr)
- p. 636 TrieNode<E> **child**(int i)
- p. 636 E **data**()
- p. 638 boolean **isDeleted**()
- p. 636 boolean **isLeaf**()
- p. 638 void **markDeleted**()
- p. 637 TrieLeafNode<E> **next**()
- p. 636 TrieNode<E> **parent**()
- p. 637 TrieLeafNode<E> **prev**()
- p. 638 void **remove**()
- p. 637 void **setNext**(TrieLeafNode<E> nextNode)
- p. 636 void **setParent**(TrieNode<E> parent)
- p. 638 void **setPrev**(TrieLeafNode<E> prevNode)

Trie.LeafNode Internal Methods

- p. 644** *LeafNode*(E data)

Trie.SearchData Public Methods

- p. 645** boolean **atLeaf**()
- p. 645** boolean **atRoot**()
- p. 648** void **moveToMaxDescendant**()
- p. 645** int **numMatches**()
- p. 647** void **retraceToLastLeftFork**(E x)

Trie.SearchData Internal Methods

- p. 644** int **childBranchPosition**(int childIndex)
- p. 645** int **childIndex**(E element)
- p. 648** void **elementsInSubtree**(Collection<? super E> c)
- p. 646** int **extendPath**(E element, TrieNode<E> newChild)
- p. 646** int **moveDown**(E element)
- p. 645** int **moveDown**(int childIndex)
- p. 646** int **moveUp**()
- p. 645** int **parentBranchPosition**()
- p. 646** boolean **processedEndOfString**(E element)

Trie.Tracker Public Methods

- p. 663** boolean **advance**()
- p. 662** E **get**()
- p. 664** boolean **hasNext**()
- p. 101 void **ignoreConcurrentModifications**(boolean ignore)
- p. 101 void **ignorePriorConcurrentModifications**()
- p. 662** boolean **inCollection**()
- p. 101 E **next**()
- p. 664** void **remove**()
- p. 663** boolean **retreat**()

Trie.Tracker Internal Methods

- p. 662** *Tracker*(TrieLeafNode<E> ptr)
- p. 101 void **checkValidity**()
- p. 663** TrieLeafNode<E> **skipRemovedElements**(TrieLeafNode<E> ptr)
- p. 101 void **updateVersion**()

Chapter 42

Compact Trie Data Structure

package collection.ordered.digitized

AbstractCollection<E> implements *Collection*<E>

└ Trie<E> implements *DigitizedOrderedCollection*<E>, *Tracked*<E>

└ CompactTrie<E> implements *DigitizedOrderedCollection*<E>, *Tracked*<E>

Uses: Java array and references

Used By: TaggedCompactTrie (Section 49.10.3)

Strengths: Significantly reduces the expected number of nodes, as compared to a trie. The reduction in the number of nodes directly implies that the space complexity of a compact trie is expected to be significantly less than that of the trie. Also, the expected height of a compact trie is less than the expected height of the trie.

Weaknesses: When many elements share a common prefix, the compact trie still requires more space than the compressed trie (Chapter 43) or Patricia trie (Chapter 44). Also, the code is slightly more complex than that for the Trie.

Critical Mutators: none

Competing Data Structures: A compressed trie (Chapter 43) can further reduce the space usage in most cases. If space efficiency is more important than time efficiency then a ternary search trie (Chapter 45) is an option to consider. Finally, if the digitizer is base 2 then a Patricia trie (Chapter 44) is a good option.

42.1 Internal Representation

Instance Variables and Constants: All of the instance variables and constants for CompactTrie are inherited from Trie.

Populated Example: [Figures 42.1](#) shows a populated examples of a compact trie.

Abstraction Function: The abstraction function is identical to that for the trie data structure. Namely, for compact trie T

$$AF(T) = seq(root).$$

42.3.1 Constructors and Factory Methods

The constructor takes *digitizer*, the digitizer to be used to define the digits for any element. It creates an empty compact trie that uses the given digitizer.

```
public CompactTrie(Digitizer<? super E> digitizer) {
    super(digitizer);
}
```

The *newInternalNode* factory method takes *o*, the data value for the new node. It creates and returns a new internal trie node holding *o*.

```
protected TrieNode<E> newInternalNode(Object o) {
    return new InternalNode();
}
```

42.3.2 Algorithmic Accessors

The primary change required from the trie data structure is the method to search for an element. In the trie data structure, if a search for element *e* ends at an internal node, *e* is a prefix of an element in the collection. If *e* is not in the collection and not a prefix of an element in the collection, a search for *e* ends at a *null* child. So when a search for *e* ends at a leaf node, either *e* is in the collection (which is the case exactly when all digits in *e* have been processed), or *e* is an extension of an element in the collection.

In a compressed trie, when a search for element *e* ends at leaf *ℓ*, it is necessary to continue to compare the digits in *e* and *ℓ.data()* to determine if *e* is in the collection. More specifically, for the branch position *bp* for *ℓ*, it is known that digits 0 to *bp* − 1 of *e* and *ℓ.data()* are the same. So these digits need not be compared again. However, starting at digit *bp*, *e* and *ℓ.data()* must be compared until either a mismatch is found, or all digits in either *e* or *ℓ.data()* are processed.

The internal *checkMatchFromLeaf* method requires that *sd* has been set by *find(e)*. It takes *e*, the target, and *sd*, a SearchData instance positioned at *ℓ*. This method continues the matching process, and returns the FindResult value for the search.

```
FindResult checkMatchFromLeaf(E e, SearchData sd) {
    E leafData = ((TrieLeafNode<E>) sd.ptr).data();
    int stop = Math.min(digitizer.numDigits(e), digitizer.numDigits(leafData));
    while (sd.numMatches < stop) { //while digits remain in both e and leafData
        int targetDigit = digitizer.getDigit(e, sd.numMatches);
        int leafDigit = digitizer.getDigit(leafData, sd.numMatches);
        int comparison = targetDigit - leafDigit;
        if (comparison < 0) { //Case 1a: e < leafData (possibly a prefix)
            if (digitizer.isPrefixFree() && targetDigit == 0)
                return FindResult.PREFIX; //prefix when end of string marker processed
            else
                return FindResult.LESS; //otherwise e less than leafData
        }
        else if (comparison > 0) //Case 1b: e > leafData
            return FindResult.GREATER;
        else { //next digit matched
            sd.numMatches++;
        }
    }
```

```

    }
    if (sd.numMatches == digitizer.numDigits(e))           //Case 2a: all digits in e were matched
        if (sd.numMatches == digitizer.numDigits(leafData))
            return FindResult.MATCHED;                    //if all digits in leafData matched then equal
        else
            return FindResult.PREFIX;                      //otherwise, e is a prefix of leafData
    else                                                     //Case 2b: all digits in leafData (but not e) processed and matched
        return FindResult.EXTENSION;                      //so e is an extension of leafData
}

```

Correctness Highlights: By the requirement of this method and `BRANCHPOSITION`, `sd.numMatches` is equal to the number of digits on the search path to the leaf node referenced by `sd.ptr`. Observe that `stop` gives the total number of digits than can be processed before reaching the end of either `e` or `leafData` (the data in the leaf where the search for `e` ended). This method will continue to compare digits of `e` and `leafData` until a mismatch is found, or no digits remain in one of the two digitized elements. The following cases may occur:

Case 1: A mismatch occurs when processing digit `d`. This case is further divided into two cases based on whether the digit of `e` or `leafData` is lexicographically smaller.

Case 1a: Digit `d` of `e` is lexicographically smaller than digit `d` of `leafData`. Observe that all prior digits of `e` and `leafData` were equal. There are two possibilities that can occur.

Case 1a(i): Digit `d` of `e` is an end of string character. This means that all of the other digits of `e` matched the corresponding digit of `leafData`. This implies that `e` is a prefix of `leafData`, in which case `PREFIX` is the correct return value.

Case 1a(ii): Digit `d` of `e` is not an end of string character. Since the first mismatch occurs where an original digit from `e` is larger than the corresponding digit from `leafData`, it follows that `e` is lexicographically smaller than `leafData`. So `LESS` is the correct return value.

Case 1b: Digit `d` of `e` is lexicographically smaller than digit `d` of `leafData`. Since the end of string character is lexicographically smaller than all other digits, the only possibility in this case is that `e` follows `leafData` in the iteration order, so `GREATER` is correctly returned.

Case 2: The last digit of `e` and/or `leafData` is processed without a mismatch. We further consider the following two subcases.

Case 2a: All digits of `e` were matched. We further divide this case into the following two subcases.

Case 2a(i): All digits of `leafData` were also processed. Thus, `e` and `leafData` have the same number of digits, all of which match. Thus `MATCHED` is properly returned.

Case 2a(ii): At least one digit in `leafData` is not processed. Thus, `leafData` has more digits than `e` with the leftmost digits in `leafData` matching those in `e`. So `PREFIX` is the correct return value.

Case 2b: All digits of `leafData` are processed without a mismatch, but at least one digit of `e` has not yet been processed. (If all digits of `e` were also processed then Case 2a would apply.) This implies that `leafData` is a prefix of `e`. So `EXTENSION` is properly returned. ■

Recall that the *find* method takes *element*, the target and *sd*, the SearchData instance to hold the ending location for the search. It returns *true* if and only if *element* is in the collection.

```

FindResult find(E element, SearchData sd){
    if (isEmpty())                                //for efficiency
        return FindResult.UNMATCHED;
    initSearchData(sd);                            //start sd at the root
    int numDigitsInElement = digitizer.numDigits(element);
    while (sd.bp < numDigitsInElement && !sd.ptr.isLeaf()) {
        if (sd.moveDown(element) == NO_CHILD)
            return FindResult.UNMATCHED;          //Case 1
    }
    if (sd.bp == numDigitsInElement)
        return sd.ptr.isLeaf()? FindResult.MATCHED : FindResult.PREFIX;    //Case 2
    else {                                          //Case 3
        sd.numMatches = sd.bp;                    //set numMatches in sd
        return checkMatchFromLeaf(element, sd);
    }
}

```

Correctness Highlights: By BRANCHPOSITION and the fact that *sd.bp* is incremented whenever *sd* is moved down one level in the compact trie ensures that *sd.ptr* = *x* exactly when the digits processed in *element* are *sp(x)*. We now discuss the cases that can occur:

Case 1: Search ends at a *null* child. By PATH, this can only occur if *element* is not in the collection. Thus *UNMATCHED* is the correct return value.

Case 2: All digits in *element* are processed without a mismatch occurring. This case is further divided into two subcases.

Case 2a: The search has reached leaf *ℓ*. By PATH it follows that *element* is equivalent to *ℓ.data()*. Thus *MATCHED* is the correct return value.

Case 2b: The search ends at an internal node *x*. By PATH, when this case occurs, *element* is a prefix of all descendants of *x*. Thus *PREFIX* is the correct return value.

Case 3: Search ends at leaf *ℓ* before all digits of *element* are processed. By BRANCHPOSITION, *sd.numMatches* is properly set completing the requirements on *sd* by *checkMatchFromLeaf*. The rest of the correctness follows from that of *checkMatchFromLeaf*. ■

42.3.3 Content Mutators

The compaction included as part of the compact trie requires changes in the methods that both add and remove elements from the collection.

Methods to Perform Insertion

We now present the changes from the trie data structure when inserting a new element into a compact trie. As with a trie, first an unsuccessful search for *element* is performed. Observe that either


```

        sd.extendPath(leaf.data(), leaf);           //put the leaf node back into the tree
        sd.moveUp();                               //reposition at the parent for the new node
    }
    sd.extendPath(element, newNode);               //Case 1, Case 2, and Case 3: add the new node
}

```

Correctness Highlights: Since the new element is not a prefix or suffix of any element currently in the collection, it follows that the search path defined by *element* either ends at a *null* child or at a leaf ℓ for which *element* is not equal to $\ell.data()$ or a suffix of $\ell.data()$. One of the following three cases must occur:

Case 1: Collection is empty. In this case, setting the root to an internal node to which the new leaf is attached (by the last line of the method) satisfies all properties.

Case 2: Search path ended at leaf node x . By PATH it follows that both *newData* and the data associated with x have the prefix $sp(x)$. The while loop begins at the parent of x , and continues adding a new internal node as long as the element in the leaf node and *element* match at the current branch position. Since *element* is guaranteed not to be a suffix of the element in the leaf, eventually a digit that differs must be reached, causing the while loop to terminate. Finally, the new node is added. Thus, PATH is preserved. The *extendPath* method preserves REACHABLE, PARENT, and NUMCHILDREN.

Case 3: Search path ended at *null* child. By the specification of *find*, in this case *sd* is at the node on the search path from which the *null* child was reached. So all that is required is to preserve PATH is to replace the *null* child with the new node. The *setChild* method preserves REACHABLE, PARENT, and NUMCHILDREN. ■

Methods to Perform Deletion

We now present the methods to remove an element from a compact trie. In order to remove a node, its index within the parent must be known. When the branch position of the parent is known, then this index can be determined in constant time using the trie node *getIndex* method. However, in both a trie and compact trie, the branch position for a node is not stored in the node since it can always be computed by the search process. In the Trie, we can easily determine branch position for a leaf without repeating the search, since the branch position is always equal to the number of digits in the element stored in the leaf. However, for a compact trie, the branch position of a leaf node cannot be determined locally. To avoid repeating the search for the leaf node to delete, our implementation instead iterates through the children of the parent to discover which child index leads to this node.

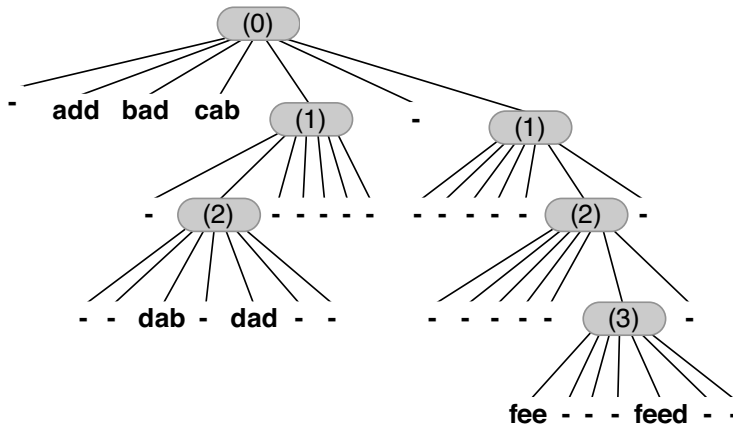
The *getIndexForChild* method that takes *parent*, a reference to the parent, and *child*, a reference to the desired child. It returns the index of *parent* the references *child*.

```

int getIndexForChild(InternalNode parent, TrieNode<E> child){
    int index = 0;
    while (parent.child(index) != child && index < childCapacity)
        index++;
    return index;
}

```

As with a trie, the leaf node ℓ holding the target element is located using *find*, and then this node is removed. By NOSINGLETONLEAF, $\ell.parent()$ has at least one other child. If $L.parent()$ has at

**Figure 42.3**

The compact trie that results when removing `ad#` and `deed#`, from the compact trie of Figure 42.1.

least two other children, no other change is required. Now consider when ℓ had a single sibling s . If s is an internal node, no other change is required. However, if s is a leaf node, to preserve `NOSINGLETONLEAF`, all internal nodes in the sequence of ancestors of ℓ that have a single child must be removed. This process ends at the first ancestor x of ℓ with some non-empty sub-trie besides that holding ℓ . Then s can be added as a child of x to replace the now empty sub-trie that used to contain ℓ .

As an example, we consider removing `ad#`, `deed#`, and then `dad#` from the compact trie shown in Figure 42.1. After removing `ad#` both its parent and grandparent have a single child. Thus both are removed making the leaf holding `add#` a child of the root. Observe that if the parent of `ad#` had two other children besides `ad#` then no internal nodes would have been removed.

To remove `deed#` the only change made is that the child reference to `deed#` is replaced by `null`. Although this leaves an internal node with a single child, this is not a violation of `NOSINGLETONLEAF`. The resulting compact trie is shown in Figure 42.3. Finally, Figure 42.4 shows the compact trie that results when `dad#` is removed causing both its parent and grandparent to also be removed.

The internal `remove` method takes `node`, a reference to the trie node to remove.

```

void remove(TrieNode<E> node) {
    ((TrieLeafNode<E>) node).remove();           //preserve OrderedLeafChain and InUse
    if (node == root) {                          //special case for a collection with one element
        root = null;
    } else {
        InternalNode parent = (InternalNode) node.parent();
        int index = getIndexForChild(parent, node);
        parent.children[index] = null;           //remove reference to node
        parent.numChildren--;                     //preserve NumChildren
        if (parent.numChildren == 1) {            //preserve NoSingletonLeaf
            TrieNode<E> onlyChild = null;        //find sibling of node (now an only child)
            for (index = 0; index < childCapacity && onlyChild == null; index++)
                if (parent.child(index) != null) //look for sibling of node
                    onlyChild = parent.child(index); //set when to other child when found
            if (onlyChild.isLeaf()) {             //can only collapse if the other child is a leaf
                InternalNode toRemove = null;
                while (parent != root && parent.numChildren == 1) { //find lowest ancestor

```

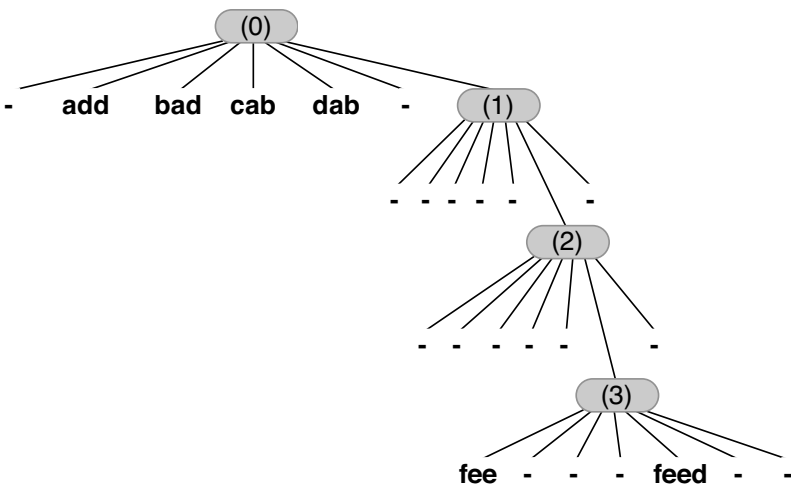


Figure 42.4
The compact trie that results when removing dad# from the compact trie of [Figure 42.1](#).

```
        toRemove = parent;                                //with only 1 child
        parent = (InternalNode) parent.parent();
    }
    index = getIndexForChild(parent, toRemove);             //replace chain of nodes with
    parent.children[index] = onlyChild;                     //one child by onlyChild
    onlyChild.setParent(parent);
}
}
}
size--;                                                    //preserve Size
}
```

Correctness Highlights: By ORDEREDLEAFCHAIN and the correctness of the TrieLeafNode *remove* method, INUSE, ORDEREDLEAFCHAIN, and REDIRECTCHAIN are preserved. The *while* loop preserves NOSINGLETONLEAF, REACHABLE, and BRANCHPOSITION. SIZE is preserved by the decrement of *size*. The remaining properties are not affected since rest of the compact trie structure is unchanged. ■

42.4 Performance Analysis

The asymptotic time complexities for all of the public methods of the CompactTrie class and Tracker class are the same as those for the trie data structures shown in Tables 41.5 and 41.6.

The most significant difference between a trie and a compact trie is in the space usage. Knuth [97] shows that the expected number of internal nodes in a compact trie is roughly $\frac{n}{\ln b}$. Thus, the expected number of references is $n \cdot \frac{b}{\ln b}$. As an illustration, observe the difference in the number of

internal nodes in the trie of Figure 41.1 and the compact trie of Figure 42.1, both which represent the same collection.

Another important difference between the trie and compact trie is the behavior during a successful search. On average, the number of child references followed during a successful search is less for the compact trie, since the length of the search path for element e is the shortest prefix of e such that no other element in the collection has that prefix. More specifically, the expected length of the search path is $O(\log_b n)$ if the elements are randomly selected (where the number of digits approaches infinity). In contrast, for the trie, the length of the search path for element e is the number of digits in e . However, observe that the number of digits examined by *checkMatchFromLeaf* are exactly those that the *Trie find* method makes in the additional calls to *moveDown*. So the number of digits examined in the *Trie* and *CompactTrie find* methods are exactly the same. Still, the reduction in the number of calls to *moveDown* does slightly reduce the time spent in *find*.

Finally, we consider the way in which the compaction performed for the compact trie affect the cost of the restructuring performed by *insert* and *remove*. Consider an element e that is being added to the compact trie, and let e have d_e digits where the first d digits is the longest prefix of e shared by some element of the collection. The number of new internal nodes added for the compact trie is $d_e - d$ less than the number of new internal nodes added for the trie. A similar savings occurs in the number of internal nodes that must be deleted when an element is removed from a compact trie, as compared to removing that same element from a trie.

42.5 Quick Method Reference

CompactTrie Public Methods

- p. 673 *CompactTrie*(Digitizer<? super E> digitizer)
- p. 98 void **accept**(Visitor<? super E> v)
- p. 659 void **add**(E element)
- p. 100 void **addAll**(Collection<? extends E> c)
- p. 659 Locator<E> **addTracked**(E element)
- p. 100 void **clear**()
- p. 656 void **completions**(E prefix, Collection<? super E> c)
- p. 97 boolean **contains**(E value)
- p. 99 void **ensureCapacity**(int capacity)
- p. 652 E **get**(int r)
- p. 96 int **getCapacity**()
- p. 96 Comparator<? super E> **getComparator**()
- p. 98 E **getEquivalentElement** (E target)
- p. 662 Locator<E> **getLocator**(E x)
- p. 96 int **getSize**()
- p. 96 boolean **isEmpty**()
- p. 661 Locator<E> **iterator**()
- p. 661 Locator<E> **iteratorAtEnd**()
- p. 657 void **longestCommonPrefix**(E prefix, Collection<? super E> c)
- p. 653 E **max**()
- p. 653 E **min**()
- p. 655 E **predecessor**(E element)
- p. 661 boolean **remove**(E element)
- p. 100 void **retainAll**(Collection<E> c)
- p. 655 E **successor**(E element)
- p. 97 Object[] **toArray**()

- p. 97 `E[] toArray(E[] array)`
- p. 98 `String toString()`
- p. 99 `void trimToSize()`

CompactTrie Internal Methods

- p. 657 `void addNewNode(TrieNode<E> newNode, SearchData sd)`
- p. 673** `FindResult checkMatchFromLeaf(E e, SearchData sd)`
- p. 97 `int compare(E e1, E e2)`
- p. 650 `SearchData createSearchData()`
- p. 97 `boolean equivalent(E e1, E e2)`
- p. 651 `FindResult find(E element, SearchData sd)`
- p. 677** `int getIndexForChild(InternalNode parent, TrieNode<E> child)`
- p. 651 `SearchData initSearchData(SearchData sd)`
- p. 658 `TrieLeafNode<E> insert(E element)`
- p. 656 `void moveToLowestCommonAncestor(E prefix, SearchData sd,`
`FindResult findStatus)`
- p. 653 `boolean moveToPred(E element, SearchData sd, FindResult findStatus)`
- p. 673** `TrieNode<E> newInternalNode(Object o)`
- p. 651 `TrieLeafNode<E> newLeafNode(E element)`
- p. 660 `void remove(TrieNode<E> node)`
- p. 661 `void removeImpl(SearchData sd)`
- p. 99 `void traverseForVisitor(Visitor<? super E> v)`
- p. 98 `void writeElements(StringBuilder s)`

Chapter 43

Compressed Trie Data Structure

package `collection.ordered.digitized`

`AbstractCollection<E>` implements *Collection<E>*

└─ `CompactTrie<E>` implements *DigitizedOrderedCollection<E>, Tracked<E>*

└─ `CompressedTrie<E>` implements *DigitizedOrderedCollection<E>, Tracked<E>*

Uses: Java array and references

Used By: `TaggedCompressedTrie` (Section 49.10.4)

Strengths: The compressed trie makes further reductions (from those of the compact trie) in both the expected number of trie nodes and also in the expected height of the trie. These savings are obtained by performing compression to remove any internal nodes with only a single child. In most cases, this compression reduces the overall space complexity of the data structure.

Weaknesses: An additional instance variable to maintain the branch position for each node must be added. Thus for a very dense collection, the space usage could be higher for the compressed trie than a compact trie. Also, the code becomes slightly more complex.

Critical Mutators: none

Competing Data Structures: If a data structure is being selected for a dense collection, a compact trie (Chapter 42) should be considered. If space efficiency is more important than time efficiency then a ternary search trie (Chapter 45) is an option to consider. Finally, if the digitizer is base 2 and the collection is naturally prefix free then a Patricia trie (Chapter 44) is a good option.

43.1 Internal Representation

The basic representation for the compressed trie is the same as that for the trie and compact trie. The significant change is that all internal nodes (versus just those whose children are leaves) have at least two children. In the trie and compact trie, the branch position for an internal node is always equal to the level of the node. This property no longer holds for the compressed trie.

For example, consider the sample tries in Figure 39.4 on page 627. Recall that the branch position is shown in parentheses in each internal node. For the trie and compact trie, the branch position is always equal to the level. However, in the compressed trie, since all elements in the collection that start with “d” have a common prefix of “da,” there is no need for an internal node along that path with a branch position of 1 that has a single child.

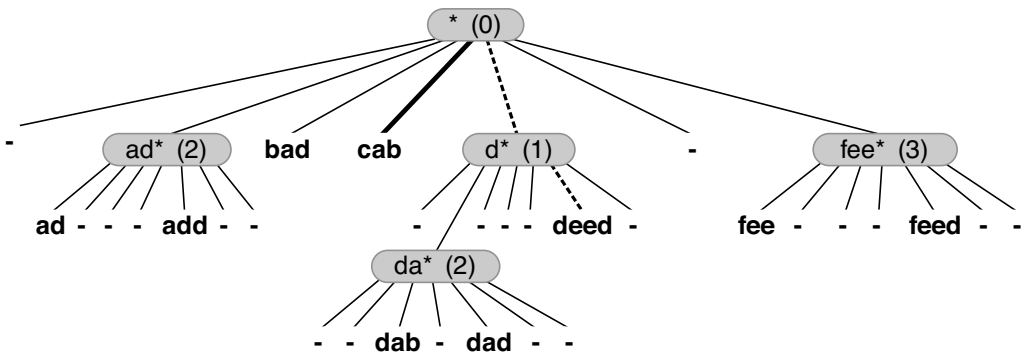


Figure 43.1
A populated example of a compressed trie holding ad#, add#, bad#, cab#, cafe#, dab#, dad#, deed#, fee#, and feed#. The structure of the trie is the same, regardless of the order in which the elements are added. The leftmost branch from each node corresponds to the end of string character #. Each leaf is visualized by showing its associated data (without the end of string character). The search path for cafe# is shown in bold, and the search path for deaf# is shown as a dashed line.

Since we cannot determine the branch position from the level, it is necessary to store the branch position as part of the internal node. We use a simple example to illustrate why adding the branch position is still not sufficient to perform a search in a compressed trie. Consider searching for dbd# in the trie of Figure 39.4. The first step in the search for dbd# goes to the internal node labeled with “da* (2).” Since the branch position is 2, digit 2 of “dbd#” determines the next branch on the search path. That digit is “d,” so the search moves to *child*(4), which is the leaf holding dad#. Since both dbd# and dad# have # for digit 3, the CompactTrie *checkMatchFromLeaf* method would incorrectly conclude that dbd# and dad# are the same. The problem is that digit 1 which was bypassed is the distinguishing digit. Also, as with the trie and compact trie, the desired behavior during a search for dbd# would be to stop the search once “db” is processed since there are no elements in the collection with the prefix of “db.”

One possible solution would be to let the search continue to a leaf (as it does for the trie and compact trie), but then compare the target to the data at the leaf starting from the leftmost digit. The drawbacks of this solution are (1) it searches further down the compressed trie than necessary in many unsuccessful searches, and (2) it generally requires repeating comparisons between digits that were made earlier in the search. To avoid these inefficiencies, we associate with each internal node the search path, which is the shared common prefix for all its descendants. The most direct implementation would let each internal node *x* have an instance variable that refers to a copy of the prefix *sp*(*x*). Instead we save space by letting the node *x* refer to leaf node *ℓ* in *T*(*x*). Let *x.data* be the data associated with *ℓ*. When a search for element *e* reaches internal node *x*, digits (*x.parent.bp*) + 1 through (*x.bp*) − 1 of *e* and *x.data* must be compared. If any mismatch occurs then *e* is not in the collection. Otherwise digit *x.bp* of *e* determines which child reference to follow. Observe that by using this approach, a search will examine exactly the prefix of the target that is needed to distinguish it from all elements currently held in the collection.

Instance Variables and Constants: All instance variables for CompressedTrie are those inherited from Trie.

Populated Example: Figures 43.1 shows a populated examples of a compressed trie. As with the trie and compact trie, the structure of a compressed trie is independent of the order in which the elements are added.

Terminology: We use the same terminology as for the Trie (Chapter 41) except that the definition for the search path must be extended for the case where the node's branch position is not necessarily one larger than that of its parent.

- We define the **search prefix** of trie node x , $sp(x)$ recursively as follows where $+$ is string concatenation, and $digitizer.getDigit(x.data, i)$ as “character” i of $x.data$. That is, in the following construction we concatenate, rather than sum, the values for each digit.

$$sp(x) = \begin{cases} \# & \text{if } x = \text{root} \\ sp(x.parent) + y & \text{otherwise} \end{cases}$$

where $+$ is the concatenation operator, and $y = \sum_{i=(x.parent.bp)+1}^{(x.bp)-1} digitizer.getDigit(x.data, i)$.

Abstraction Function: The abstraction function is the same as that for the trie and compact trie. Namely, for compressed trie T

$$AF(T) = seq(root).$$

Optimizations: If desired, it is possible to reduce the space usage, at the cost of increasing the time complexity, by not referencing the associated data at the internal nodes. However, then each search must continue to a leaf, and then compare the digits between the target and the data at the leaf starting at the leftmost digit until it is determined if they are equivalent.

When an internal trie node references an element, it retains that element in memory for comparison purposes even if the application has removed the element from the trie. Consequently, these elements do not become eligible for garbage collection. A space-saving optimization, at the cost of additional time during remove, would be to trace back up the tree to the root, replacing in each internal node any reference to the removed node by its sibling. This preserves all properties of the trie, while purging it of any lingering references to the removed element. As a further optimization, one could remember whether or not the target element was encountered in an internal node during the search for the element being removed, and make the purging pass up the tree only in such cases.

43.2 Representation Properties

We inherit all of the Trie properties except for `BRANCHPOSITION`, which is overridden. We also add the `NOSINGLETONNODE` property.

BRANCHPOSITION: For each compressed trie node x , $x.bp$ is equal to the number of characters in $sp(x)$. Thus for each `LeafNode` ℓ , $\ell.bp$ is the number of characters in the longest common prefix shared by all leaves in $T(x)$.

NOSINGLETONNODE: Every node (except the root) has a sibling.

43.3 Compressed Trie Node Interface

TrieNode<E>

└ ***CompressedTrieNode***<E>

Along with the methods inherited from the *TrieNode* interface, the following method is added to the *CompressedTrieNode* interface.

int bp(): Returns the position of the digit used for branching at this node.

43.4 Internal Node Inner Class

AbstractTrieNode<E> implements *TrieNode*<E>

└ *TrieNode*<E>.InternalNode implements *TrieNode*<E>

└ ***CompressedTrieNode***<E>.InternalNode implements ***CompressedTrieNode***<E>

In this section, we describe the *InternalNode* inner class of *CompressedTrieNode*. It extends the *InternalNode* inner class of the *Trie* node by adding an instance variable *bp* to store the branch position, and a reference *data* to the leaf node used for the associated data. Since *PatriciaTrie* that extends *CompressedTrie* will directly store an element providing the shared prefix, the type for *data* is any object.

```
int bp;                                //branch position
E data;                                //reference to the associated data
```

We introduce a constructor that takes *bp*, the branch position for the internal node, and *data*, the leaf node to reference for the data.

```
InternalNode(int bp, E data) {
    this.bp = bp;
    this.data = data;
}
```

The *data* method returns the data associated with this internal node.

```
public E data() {
    return data;
}
```

The *bp* method returns the branch position for this internal node.

```
public int bp() {
    return bp;
}
```

Correctness Highlights: Follows from BRANCHPOSITION. ■

Finally, the *setChild* takes *child*, a reference to a trie node. It adds *child* as a child of this node.

```
protected void setChild(TrieNode<E> child) {
    setChild(child, child.data(), bp);
}
```

Correctness Highlights: By BRANCHPOSITION, *bp* is the branch position for this node. The rest of the correctness follows from that of *setChild*. ■

43.5 Compressed Trie Leaf Node Inner Class

AbstractTrieLeafNode<E> implements *TrieLeafNode*<E>

└ Trie<E>.LeafNode implements *TrieNode*<E>, *TrieLeafNode*<E>

└ LeafNode implements ***CompressedTrieNode***<E>, ***TrieLeafNode***<E>

In this section, we describe the LeafNode inner class that extends LeafNode from Trie. Recall that for an internal trie node x , $x.bp$ is the largest value such that all descendants of $T(x)$ share a common prefix of $x.bp$ digits. To make the code more uniform, it is convenient to use the same definition to define the branch position for a leaf node. Since a leaf node ℓ is the only element in $T(\ell)$, it follows that the branch position for a leaf node should be the number of digits in its associated data.

```
LeafNode(E data) {
    super(data);
}
```

The *bp* method returns the branch position for this LeafNode.

```
public int bp() {
    return digitizer.numDigits(data());
}
```

43.6 Compressed Trie Search Data Inner Class

SearchData

└ ***CompressedTrieSearchData***

We override the SearchData inner class of Trie to use the *branchPosition* instance variable to determine the branch position for an internal node. The *childBranchPosition* take *childIndex*, the index for the desired child, and returns the branch position for the specified child of this node.

```

int childBranchPosition(int childIndex){
    return ((CompressedTrieNode<E>) ptr.child(childIndex)).bp();
}

```

The *parentBranchPosition* returns the branch position for the parent of this node.

```

int parentBranchPosition(){
    return ((CompressedTrieNode<E>) ptr.parent()).bp();
}

```

Unlike in the trie and compact trie, the number of matches in the initial search to the leaf may not equal the *branchPosition* for a compressed trie node on the search path. Therefore, we modify the *numMatches* method to return the number of digits matched so far during a search.

```

public int numMatches() {
    return numMatches;
}

```

43.7 Compressed Trie Methods

In this section, we present the methods for the *CompressedTrie* class.

43.7.1 Constructors and Factory Methods

The constructor takes *digitizer*, the digitizer to be used to define the digits for any element. It creates an empty compressed trie that uses the given digitizer.

```

public CompressedTrie(Digitizer<? super E> digitizer) {
    super(digitizer);
}

```

The *createSearchData* factory method returns a newly created and initialized compressed trie search data instance.

```

protected CompressedTrieSearchData createSearchData() {
    return initSearchData(new CompressedTrieSearchData());
}

```

Recall that *initSearchData* takes *sd*, the *SearchData* instance to initialize to the root, and returns the initialized *SearchData* instance.

```

CompressedTrieSearchData initSearchData(SearchData sd){
    sd.ptr = root;
    sd.numMatches = 0;
    sd.bp = 0;
    if (root != null)
        ((CompressedTrieSearchData) sd).bp =
            ((CompressedTrieNode<E>) root).bp();
    return (CompressedTrieSearchData) sd;
}

```

//place sd at root
//initially no matches
//root has branch position 0
//only use sd.branchPosition when root isn't null

The *newLeafNode* factory method takes *element*, the data element, and returns a newly created compressed trie leaf node holding the given element.

```
TrieLeafNode<E> newLeafNode(E element){
    return new LeafNode(element);
}
```

The *newInternalNode* factory method takes *data*, the associated data for the new node. It creates and returns a new internal compressed trie node with the specified branch position and data.

```
TrieNode<E> newInternalNode(int bp, E data) {
    return new InternalNode(bp, data);
}
```

43.7.2 Algorithmic Accessors

Recall that the *find* method takes *target*, the target element, and returns a reference to a leaf node that contains an occurrence of the target if it is found. Otherwise it returns a reference to the internal node where the search ends. Unlike the trie and compact trie that always compare one digit per internal node, the internal compression performed by the compressed trie requires multiple comparisons within an internal node. If a mismatch occurs during one of these comparisons, the search ends with a return value of *UNMATCHED*.

```
FindResult find(E target, SearchData sd){
    if (isEmpty()) //for efficiency
        return FindResult.UNMATCHED;
    initSearchData(sd);
    CompressedTrieSearchData ctsd = (CompressedTrieSearchData) sd; //reduces casting
    int digitsInTarget = digitizer.numDigits(target); //last digit position to check
    for (int d = 0; d < digitsInTarget && !ctsd.atLeaf(); d++, ctsd.numMatches++) {
        if (d == ctsd.bp) { //branch at the current node
            if (ctsd.moveDown(target) == NO_CHILD) //move to next node
                return FindResult.UNMATCHED; //no match if desired child is null
            else { //compare digit d of target and the associated data of current trie node
                int comparison = digitizer.getDigit(target, d)
                    - digitizer.getDigit(ctsd.ptr.data(), d);
                if (comparison < 0) //when digit d of target < digit d of data for current node
                    return FindResult.LESS; //target < data of current node
                else if (comparison > 0)
                    return FindResult.GREATER; //target > data of current node
            }
        }
    }
    if (ctsd.numMatches == digitsInTarget && !ctsd.atLeaf())
        return FindResult.PREFIX; //processed all digits in target but at internal node
    else
        return checkMatchFromLeaf(target, ctsd); //otherwise check remaining digits
}
```

Correctness Highlights: By PATH we are guaranteed that if $pos = ctsd.branchPosition$ that the prefix of *target* processed so far is the search path for the current node. We now consider all the cases in which the *for* loop can be exited.

Case 1: Search ends at a null child. By PATH this can only occur if the target is not in the collection. Thus *UNMATCHED* is the correct return value.

Case 2: Search ends with a mismatch at a digit d for which $pos \neq ctsd.branchPosition$. From PATH it also follows that when the digit d of *target* is not equal to the data associated with the current internal node, then there is no equivalent element in the collection. Since the first $d - 1$ digits have matches, if digit d of the target is less than the digit d of the data associated with the current internal node, *LESS* is the correct return value. Similarly, if digit d of the target is greater than the digit d of the data associated with the current internal node, *GREATER* is the correct return value

Case 3: All digits in *target* have matched, yet *sd* is at an internal node. By PATH it follows that *target* is a prefix of all the elements in the subtree rooted at the current internal node.

Case 4: All digits in *target* have matched and the current search data position is at a leaf. This case reduces to the same situation as in the compact trie. The correctness follows from the inherited *checkMatchFromLeaf* method of the compact trie.

■

43.7.3 Content Mutators

The compaction behavior of the compressed trie requires changes in the methods that add and remove elements from the collection.

Methods to Perform Insertion

We now present the changes from the compact trie data structure when inserting a new element into a compressed trie. Let x be the search location after an unsuccessful search for *element*. By PATH, all descendants of x and *element* share a common prefix of $sp(x)$. Unlike the compact trie in which a sequence of new internal nodes might be needed, for the compressed trie at most one new internal node must be added. As with the compact trie, when the search ends at a *null* child, the new element can just be added as a child of *sd.ptr*. We now consider the case in which the search either ends at a leaf node or with a mismatch at an internal node.

For example, consider inserting *cafe#* into the compressed trie of [Figure 43.1](#). (The search path for *cafe#* is shown in bold.) The search ends at leaf ℓ holding *cab#* with two digits having matched the target. Thus a new internal node x with a branch position of 2 replaces ℓ and then ℓ and *cafe#* are added as leaves. Observe that $sp(x) = \text{"ca."}$ The resulting compressed trie is shown in [Figure 43.2](#). As with the compact trie, *cafe#* is inserted in the ordered leaf chain between *cab#* and *dab#*.

The *addNewNode* method takes *newNode*, a trie node to add that already holds the new element and *sd*, a SearchData instance. It requires that *sd* has been set by a call to *find* using the element associated with *newNode*, and that the new element will preserve the prefix free requirement for the collection.

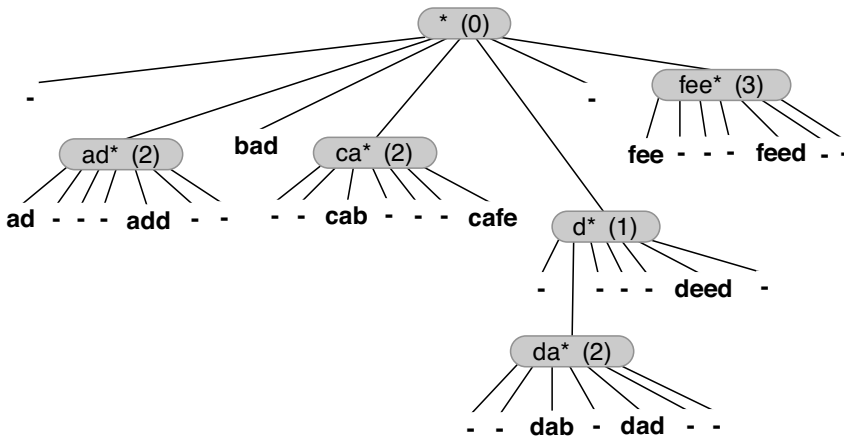


Figure 43.2
The trie that results when inserting cafe# into the compressed trie shown in Figure 43.1.

```
protected void addNewNode(TrieNode<E> newNode, SearchData sd) {
    if (isEmpty()) { //Case 1: the collection is empty
        root = newNode; //the root should directly reference the new node
    } else {
        CompressedTrieNode<E> searchLoc = (CompressedTrieNode<E>) sd.ptr;
        if (sd.atLeaf() || sd.numMatches() < searchLoc.bp()) { //Cases 2 and 3
            InternalNode node =
                (InternalNode) newInternalNode(sd.numMatches(), newNode.data());
            InternalNode searchLocParent = (InternalNode) searchLoc.parent();
            if (searchLocParent == null) //preserve Parent property
                root = node;
            else
                searchLocParent.setChild(node);
            node.setChild(searchLoc); //add old leaf as one child
            node.setChild(newNode); //add the new node as the other child
        } else { //Case 4: search ended at a null child
            ((InternalNode) searchLoc).setChild(newNode); //just add the new node
        }
    }
    sd.ptr = newNode; //set search location to the new node
}
```

Correctness Highlights: Since the new element is not a prefix or suffix of any element currently in the collection, it follows that the collection is currently empty, or the search path defined by *element* either ends at a leaf *ℓ*, at an internal node *x* where a mismatch occurs before digit *x.bp* is processed, or at a *null* child. One of the following four cases must occur:

- Case 1: Collection is empty.** In this case, setting the root to the new node satisfies all properties.
- Case 2: Search path ends at leaf *ℓ*.** For example, consider inserting deaf# into the compressed trie of Figure 43.2. In this case, a new *InternalNode node* with a branch position of

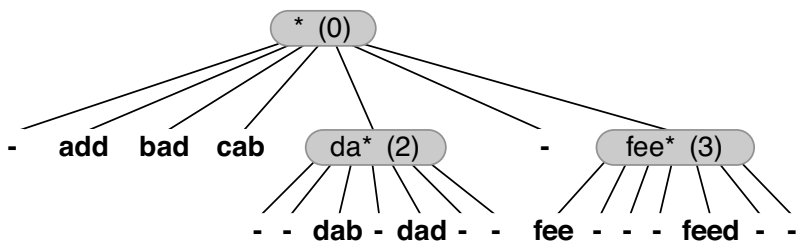


Figure 43.3
The compressed trie that results when removing ad# and deed#, from the compressed trie shown in Figure 43.1.

sd.numMatches replaces ℓ . By the requirement on this method, *element* is not equal to $\ell.data()$ or a suffix of $\ell.data()$, guaranteeing that they must differ at some digit. Setting *node.bp* to the number of common digits between *L.data* and *newNode.data* preserves BRANCHPOSITION. Since both ℓ and *newNode* are placed as children of *node*, NOSINGLETONNODE is preserved. The rest of the correctness follows from requirements on this method, and the correctness of the *setChild* method.

Case 3: Search path ends at an internal node x where a mismatch occurred. This case occurs exactly when the number of matches that have been made is less than $x.bp$. An example occurrence of this case would be if fad# were inserted into the compressed trie of Figure 43.2. As in Case 2, a new InternalNode *node* with a branch position of *sd.numMatches* replaces x . Again, x and *newNode* are placed as children of *node*. The rest of the correctness is the same as in Case 2.

Case 4: Search path ends at a null child. By the specification of *find*, in this case *sd* is at the node on the search path from which the null child was reached. Observe that the number of matches that occurred during the search is equal to $x.bp$, which is how it is distinguished from Case 3. For example, consider inserting cad# into the compressed trie of Figure 43.2. By just adding *newNode* as a child of x (which already has at least two children), NOSINGLETONNODE is preserved. The *setChild* method preserves REACHABLE, PARENT, and NUMCHILDREN.

Finally, so that the predecessor can be efficiently found by the inherited *insert* method without a new search, *sd.ptr* is updated to reference *newNode*. It is not necessary to update *numMatches* since it is not used after this point in the insertion procedure.

Methods to Perform Deletion

We now consider removing an element from a compressed trie. As with the compact trie, first the leaf node ℓ holding the element to remove is located using *find*, and then the reference to ℓ is replaced by null. Then starting at ℓ 's parent and moving up to the root, any internal node with a single child is removed. Then the single remaining sibling of ℓ is added as a child of the deepest ancestor not removed.

As an example, consider removing ad#, deed#, and then dad# from the compressed trie shown in Figure 42.1. After removing ad#, its parent has a single child and so is removed making add# a child of the root.

When deed# is removed, its parent x has a single child y (the internal node labeled by “da*

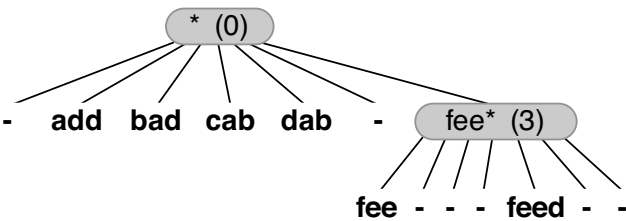


Figure 43.4
The trie that results when removing dad# from the compressed trie shown in [Figure 43.1](#).

(2)"). Thus, x is removed, making y a child of the root. The resulting compressed trie is shown in [Figure 43.3](#). Finally, Figure 43.4 shows the compressed trie that results when dad# is removed, causing its parent also to be removed.

```
protected void remove(TrieNode<E> node) {
    ((LeafNode) node).remove(); //preserve OrderedLeafChain
    InternalNode parent = (InternalNode) node.parent();
    int childIndex = digitizer.getDigit(node.data(), parent.bp());
    parent.children[childIndex] = null; //remove node from the trie
    parent.numChildren--; //preserve NumChildren
    if (parent.numChildren == 1) { //preserve NoSingletonNode
        TrieNode<E> sibling = null; //find only sibling
        for (int i = 0; i < childCapacity && sibling == null; i++)
            if (parent.child(i) != null)
                sibling = parent.child(i);
        if (parent == root) //if root was node with single child
            root = sibling;
        else //splice out node with a single child
            ((InternalNode) parent.parent()).setChild(sibling);
    }
    size--; //preserve Size
}
```

Correctness Highlights: The comments within the code show where most of the properties are maintained. The remaining properties are maintained by the *setChild* method. ■

43.8 Performance Analysis

The time complexity for the compressed trie is the same as for the trie and compact trie (see Tables 41.5 and 41.6). While a compressed trie uses fewer nodes than a compact trie, two additional instance variables are added for each internal node. However, the reduction in the expected number of nodes in the compressed trie slightly improves the cost of insertion since fewer new internal nodes must be allocated and added to the structure. Similar savings occur when removing an element.

43.9 Quick Method Reference

CompressedTrie Public Methods

- p. 688 *CompressedTrie*(Digitizer<? super E> digitizer)
- p. 98 void **accept**(Visitor<? super E> v)
- p. 659 void **add**(E element)
- p. 100 void **addAll**(Collection<? extends E> c)
- p. 659 Locator<E> **addTracked**(E element)
- p. 100 void **clear**()
- p. 656 void **completions**(E prefix, Collection<? super E> c)
- p. 97 boolean **contains**(E value)
- p. 99 void **ensureCapacity**(int capacity)
- p. 652 E **get**(int r)
- p. 96 int **getCapacity**()
- p. 96 Comparator<? super E> **getComparator**()
- p. 98 E **getEquivalentElement** (E target)
- p. 662 Locator<E> **getLocator**(E x)
- p. 96 int **getSize**()
- p. 96 boolean **isEmpty**()
- p. 661 Locator<E> **iterator**()
- p. 661 Locator<E> **iteratorAtEnd**()
- p. 657 void **longestCommonPrefix**(E prefix, Collection<? super E> c)
- p. 653 E **max**()
- p. 653 E **min**()
- p. 655 E **predecessor**(E element)
- p. 661 boolean **remove**(E element)
- p. 100 void **retainAll**(Collection<E> c)
- p. 655 E **successor**(E element)
- p. 97 Object[] **toArray**()
- p. 97 E[] **toArray**(E[] array)
- p. 98 String **toString**()
- p. 99 void **trimToSize**()

CompressedTrie Internal Methods

- p. 657 void **addNewNode**(TrieNode<E> newNode, SearchData sd)
- p. 673 FindResult **checkMatchFromLeaf**(E e, SearchData sd)
- p. 97 int **compare**(E e1, E e2)
- p. 650 SearchData **createSearchData**()
- p. 97 boolean **equivalent**(E e1, E e2)
- p. 651 FindResult **find**(E element, SearchData sd)
- p. 689 FindResult **find**(E target, SearchData sd)
- p. 677 int **getIndexForChild**(InternalNode parent, TrieNode<E> child)
- p. 651 SearchData **initSearchData**(SearchData sd)
- p. 658 TrieLeafNode<E> **insert**(E element)
- p. 656 void **moveToLowestCommonAncestor**(E prefix, SearchData sd, FindResult findStatus)
- p. 653 boolean **moveToPred**(E element, SearchData sd, FindResult findStatus)
- p. 673 TrieNode<E> **newInternalNode**(Object o)
- p. 689 TrieNode<E> **newInternalNode**(int bp, E data)
- p. 651 TrieLeafNode<E> **newLeafNode**(E element)
- p. 660 void **remove**(TrieNode<E> node)
- p. 661 void **removeImpl**(SearchData sd)
- p. 99 void **traverseForVisitor**(Visitor<? super E> v)
- p. 98 void **writeElements**(StringBuilder s)

CompressedTrie.CompressedTrieSearchData Public Methods

p. 688 `int numMatches()`

CompressedTrie.CompressedTrieSearchData Internal Methods

p. 687 `int childBranchPosition(int childIndex)`

p. 688 `int parentBranchPosition()`

CompressedTrie.InternalNode Public Methods

p. 686 `int bp()`

p. 636 `TrieNode<E> child(int i)`

p. 643 `int childIndex(E element, int bp)`

p. 636 `E data()`

p. 636 `boolean isLeaf()`

p. 636 `TrieNode<E> parent()`

p. 636 `void setParent(TrieNode<E> parent)`

CompressedTrie.InternalNode Internal Methods

p. 686 `InternalNode(int bp, E data)`

p. 687 `void setChild(TrieNode<E> child)`

p. 643 `int setChild(TrieNode<E> child, E element, int bp)`

CompressedTrie.LeafNode Public Methods

p. 638 `void addAfter(TrieLeafNode<E> ptr)`

p. 687 `int bp()`

p. 636 `TrieNode<E> child(int i)`

p. 636 `E data()`

p. 638 `boolean isDeleted()`

p. 636 `boolean isLeaf()`

p. 638 `void markDeleted()`

p. 637 `TrieLeafNode<E> next()`

p. 636 `TrieNode<E> parent()`

p. 637 `TrieLeafNode<E> prev()`

p. 638 `void remove()`

p. 637 `void setNext(TrieLeafNode<E> nextNode)`

p. 636 `void setParent(TrieNode<E> parent)`

p. 638 `void setPrev(TrieLeafNode<E> prevNode)`

CompressedTrie.LeafNode Internal Methods

p. 687 `LeafNode(E data)`

Chapter 44

Patricia Trie Data Structure

package `collection.ordered.digitized`

`AbstractCollection<E>` implements *Collection<E>*

└ `Trie<E>` implements *DigitizedOrderedCollection<E>, Tracked<E>*

└ `CompactTrie<E>` implements *DigitizedOrderedCollection<E>, Tracked<E>*

└ `CompressedTrie<E>` implements *DigitizedOrderedCollection<E>, Tracked<E>*

└ **`PatriciaTrie<E>`** implements ***DigitizedOrderedCollection<E>, Tracked<E>***

Uses: Java array and references

Used By: `TaggedPatriciaTrie` (Section 49.10.5)

Strengths: Reduces the space requirement of the compressed trie by letting each internal node also serve as a leaf node, hence removing the need to allocate leaf nodes.

Weaknesses: Only applicable when there are exactly two characters in the alphabet (and no end of string character can be used). Thus, the prefix-free digitizer cannot be used and the collection must be naturally prefix free.

Critical Mutators: none

Competing Data Structures: If the alphabet size is greater than two, or if an element in the collection would be a prefix of another (without including an end of string character) then the compressed trie (Chapter 43) should be used when time complexity is the primary concern. When it is more important to reduce the space usage, a ternary search trie (Chapter 45) should be considered.

44.1 Internal Representation

The basic representation for a Patricia trie is like that of the compressed trie, except for key modifications. First, observe that in a binary tree in which each internal node has two children (called a **full binary tree**), the number of leaf nodes is one larger than the number of internal nodes. By introducing a “sentinel” root node that can also serve as one of the leaf nodes, each internal node can also serve as a leaf node. So, instead of creating an explicit leaf node the last reference on a search path can be made to a Patricia trie node that serves as both a leaf node (i.e., the ordered leaf chain is maintained by them) and an internal node (holding the branch position, associated data, and children references). The *root* references a sentinel root, and the left child of the sentinel root references to true root. The right child of the sentinel root node is always *null*, and the sentinel root

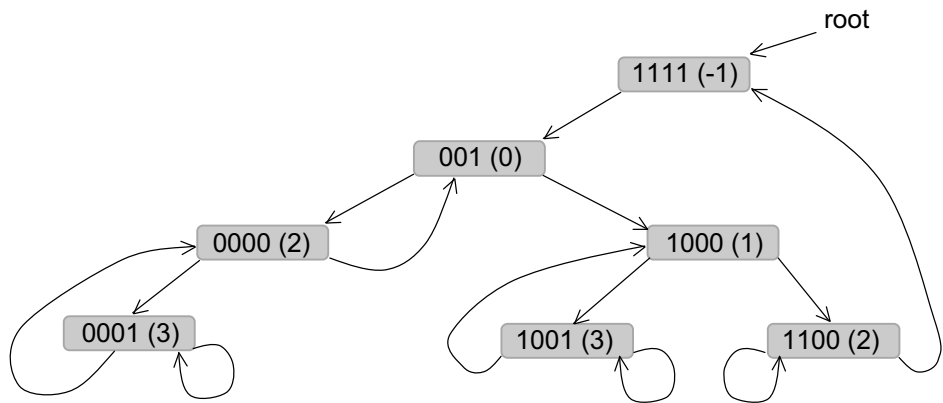


Figure 44.1
The underlying representation (excluding the ordered leaf chain) for a Patricia trie holding {1111, 1000, 001, 1001, 1100, 0000, 0001}.

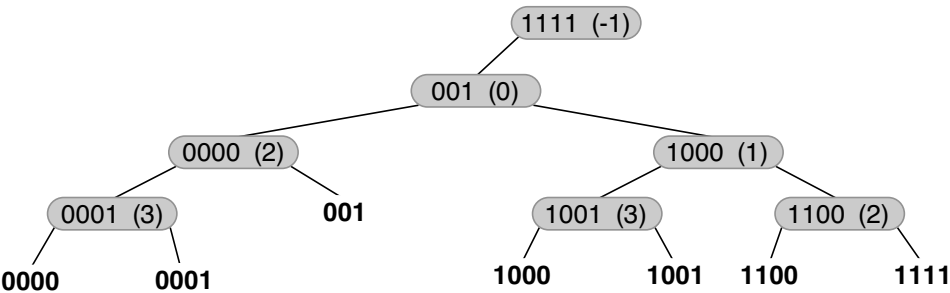


Figure 44.2
An alternate view of the Patricia trie of Figure 44.1 in which the arrows pointing upwards to a leaf are replaced by explicitly showing a leaf node. We use this view throughout the chapter.

node is given a branching factor of -1. In a Patricia trie, the only *null* child reference is *root.child(1)*. Figure 44.1 shows the underlying representation for a Patricia trie, omitting the ordered leaf chain.

In general, the Patricia trie methods are best understood when viewing the Patricia trie as shown in Figure 44.2, an alternate view for the Patricia trie of Figure 44.1. This view essentially draws each Patricia trie as a compressed trie by showing each node twice: once as an internal node and once as a separate leaf node. By viewing a Patricia trie in this manner, the similarity between the compressed trie and Patricia trie methods are more clear. The primary complication in the Patricia trie methods, caused by the internal representation not having explicit leaf nodes, is in keeping track of when the current node in the search path should be treated as an internal node and when it should be treated as a leaf node. The key observation used is that if the branch position of the current node on the search path is smaller than that of the one that preceded it, then a leaf node has been reached.

Instance Variables and Constants: All instance variables for PatriciaTrie are those inherited from the CompressedTrie.

Populated Example: Figure 44.1 shows the underlying representation for a Patricia trie. Figure 44.2 illustrates this same Patricia trie by replacing each reference upward by a reference to the implicit leaf element.

Terminology: We use the same terminology as for the trie (Chapter 41), except that as done for the compressed trie (Chapter 43), the definition of the search prefix must be modified. The changes made are an extension of that given for the compressed trie that is adjusted to account for the sentinel root. We also introduce some additional terminology.

- We refer to $root.child(0)$ as the **search root** since it is the starting point for each search.
- We define the **search prefix** of trie node x , $sp(x)$ recursively as follows:

$$sp(x) = \begin{cases} \epsilon, & \text{if } x = root.child(0) \\ sp(x.parent) + y, & \text{otherwise} \end{cases}$$

where $+$ is concatenation, and $y = \sum_{i=(x.parent.bp)+1}^{(x.bp)-1} digitizer.getDigit(x.data, i)$.

- Each Patricia trie node may simultaneously serve two roles, as a **leaf node** and as an **internal node**.
- When we reach a node during a search or traversal, we visit it in one of these two roles depending upon the way in which we have reached the node. We say that a search data instance sd is **at a leaf** if $sd.ptr.bp < cameFrom.bp$ where $cameFrom$ is the prior node on the search path.
- When we are visiting a Patricia trie node as a leaf node we say that the search data instance sd is **at a leaf node**, and when we are visiting it as an internal node we say sd is **at an internal node**.
- We say that $node.child(i)$ **references a leaf node** if $node.bp \geq node.child(i).bp$. In such cases, we say that $node$ is the **leaf parent** of $node.child(i)$.
- We say that $node.child(i)$ **references an internal node** if $node.bp < node.child(i).bp$. In such cases, we say that $node$ is the **internal node parent** of $node.child(i)$.

Abstraction Function: The Patricia trie abstraction function is the same as that for the trie, compact trie, and compressed trie. Namely, for Patricia trie T

$$AF(T) = seq(root).$$

Design Notes: Since each Patricia trie node serves as both an internal node and leaf node, it must inherit the functionality of both of these classes. However, Java does not support multiple inheritance. We resolve this difficulty by using the **adaptor design pattern** to enable the Patricia trie node to share behaviors of the compressed trie internal nodes and leaf nodes. More specifically, since most of the non-shared methods involve the ordered leaf chain, we have the Patricia trie node extend the `LeafNode` of `CompressedTrie`. Then any methods that are not shared by the compressed trie internal node and leaf node, are reintroduced for the Patricia trie node.

44.2 Representation Properties

We inherit most of the properties from the compressed trie. However, some properties must be modified to compensate for the fact that each Patricia trie node serves as both an internal node and

a leaf node. `BRANCHPOSITION` is adapted so that `bp` stores the branch position for each node in its role as an internal node. Similarly, `PARENT` must be adapted to account for the sentinel root and also the fact that the parent when viewing a Patricia trie node as an internal node, is different from its parent when it is viewed as a leaf node. The actual `parent` and `child` references will always be based on treating the node as an internal node. Also, recall that for the sentinel root, `child(0)` always references the actual root and `root.child(1)` is `null`. We also explicitly include the property stating that `root.child(1)` is the only `null` child.

Since `x.parent` is always `x`'s internal node parent, a mechanism is needed to determine `x`'s leaf parent. We achieve this by introducing a `SearchData` instance variable `cameFrom` that is the previous node on the search path. Since `cameFrom` is the node that precedes `sd` on the search path, when `sd` is at node `x` in its leaf role, `sd.cameFrom` is the leaf parent of `x`.

The correctness of the inherited compressed trie methods depend upon the fact that the associated data for a compressed trie internal node `x` is a reference to some leaf node in $T(x)$. `DESCENDANTOFSELF` preserves this property by enforcing that each node in its leaf role is a descendant of itself in its role as an internal node.

BRANCHPOSITION: For each Patricia trie Node `x` except the sentinel root, `x.bp` is equal to the number of characters in `sp(x)` for `x`. For the sentinel root, `root.bp = -1`.

PARENT: With the exception that `root.child(1) = null`, for all reachable nodes `x` and $i \in \{0, 1\}$, `x.child(i).parent = x`. Observe that this implies that with the exception of `root.child(1)`, there are no `null` child references.

DESCENDANTOFSELF: Node `x` in its leaf role is a descendant of node `x` in its role as an internal node.

44.3 Patricia Trie Node Inner Class

`Trie<E>.LeafNode` implements `TrieNode<E>`, `TrieLeafNode<E>`

└ `CompressedTrie<E>.LeafNode` implements `CompressedTrieNode<E>`, `TrieLeafNode<E>`

└ **Node** implements **`CompressedTrieNode<E>`**, **`TrieLeafNode<E>`**

In this section, we describe the `Node` inner class of `PatriciaTrie` that extends the `LeafNode` of `CompressedTrie`. It must implement both the `CompressedTrieNode` and `TrieLeafNode` interfaces. So that each Patricia trie node can also serve as an internal node, we must add instance variables for the array of children references and the branch position.

```
int bp;                                //branch position
TrieNode<E>[] children;                 //array of child references
```

The `Node` constructor takes `bp`, the branch position for this node, and `data`, the data for this node.

```
Node(int bp, E data) {
    super(data);                        //call the compressed trie LeafNode constructor
    this.bp = bp;                       //save the branch position
    children = new TrieNode[childCapacity]; //allocate the children array
}
```


We inherit methods from the leaf node class, but must define the additional methods required by the internal node interface. Recall the *bp* method returns the branch position for the node.

```
public int bp() {
    return bp;
}
```

The *child* method takes *i*, the index for the desired child, and returns the *i*th child. It throws an *IllegalArgumentException* when *i* is not between 0 and *childCapacity* - 1 (inclusive). This method is exactly as it was for the trie, compact trie, and compressed trie.

```
public TrieNode<E> child(int i){
    if (i < 0 || i ≥ childCapacity)
        throw new IllegalArgumentException();
    return children[i];
}
```

The *oppositeNode* method takes *element*, the target. It returns the child that is *not* on the search path to *element*.

```
Node oppositeNode(E element) {
    int index = childIndex(element, bp);
    return (Node) child(1-index);
}
```

Recall the *childIndex* method takes *element*, the element for which the child index sought, and *bp*, the branch position of this node. It returns the index of the child determined by *sp(element)* when this node has branch position *bp*. This method is like the one for the trie internal node, except that when *bp* is -1 (indicating that the node is the sentinel root) then the left child (child 0) is returned. Otherwise, digit *bp* of *element* determines the index of the child.

```
public int childIndex(E element, int bp){
    if (bp == -1) return 0;
    else return digitizer.getDigit(element, bp);
}
```

When a child is added to a node, its position is determined with respect to a particular element (which defines a search path) and a branch position (on that search path). In particular, the *setChild* method takes *child*, the new child to add, *element*, the element defining the search path, and *bp*, the branch position of this node. This method gets the appropriate child index for the give element and branch position and places the given child at that index. This method returns the index at which the child is placed.

```
int setChild(TrieNode<E> child, E element, int bp) {
    int i = childIndex(element, bp);
    children[i] = child;
    if (child == root || ((CompressedTrieNode<E>) child).bp() > bp)
        child.setParent(this);
    return i;
}
```

Correctness Highlights: This is just like the *setChild* method for the Trie InternalNode, except that the *setParent* method is called only if the child is the sentinel root (which serves as an internal

node) or if the child is an internal node (which occurs exactly when its branch position is smaller than that of its parent). ■

Another *setChild* method takes a single parameter *child*, the child to add below this node. This method is exactly the same as the one used for the compressed trie internal node. It must be duplicated here since the Patricia trie node extends the leaf node.

```
void setChild(TrieNode<E> child) {
    setChild(child, child.data(), bp);
}
```

44.4 Patricia Trie Search Data Inner Class

SearchData

```

└─ CompressedTrieSearchData
    └─ PatriciaSearchData
```

We extend the *SearchData* inner class of *CompressedTrie* to maintain the prior node on the search path. As discussed at the start of this chapter, using this information is the only way to find the leaf parent of a node, and is also needed to determine if *sd.ptr* should be treated as an internal node or a leaf node.

All Patricia search data methods maintain that if the search data instance *sd* is at a leaf node, then *cameFrom* is the previous node on the search path defined by *sp(ptr.data)*. In particular, when *sd* is at a node *x* in its role as a leaf, then *sd.cameFrom* references the leaf parent of *x*.

Node cameFrom;

Recall the *atLeaf* method returns *true* if and only if the search data object is currently at a leaf node.

```
public boolean atLeaf(){
    return ((Node) ptr).bp() ≤ ((Node) cameFrom).bp();
}
```

Correctness Highlights: Recall that *cameFrom* is the node that precedes *ptr* on the search path. Thus, by *BRANCHPOSITION*, this search location is at a leaf exactly when its branch position is less than that of the previous node on the search path. ■

The *atRoot* method returns *true* if and only if the *SearchData* location is at the search root in its role as an internal node.

```
public boolean atRoot() {
    return cameFrom == root;
}
```

Correctness Highlights: Recall that the search root of the Patricia trie is the left child of the sentinel root. However, just comparing *ptr* to *root.left(0)* is not sufficient since *root.left(0)* also

serves as a leaf. Since *cameFrom* is the node that precedes *ptr* on the search path, it follows that *ptr* is at the search root exactly when the node that preceded it is the sentinel root. ■

The *moveDown* method that takes *childIndex*, the index of the child to which to move. It returns the index for convenience. This method requires that *childIndex* is not 1 if *ptr* = *root*.

```
protected int moveDown(int childIndex){
    cameFrom = (Node) ptr;
    ptr = ptr.child(childIndex);
    bp = ((CompressedTrieNode<E>) ptr).bp();
    return childIndex;
}
```

Correctness Highlights: This method is very similar to that of *CompressedTrie* (which is inherited from *Trie*). There are two major differences. First by *NONULLCHILD* and the requirement on this method, we are guaranteed that after the update *ptr* will not be *null*. Setting *cameFrom* to the current position prior to the update preserves *CAMEFROM*. The rest of the correctness follows from *PATH*. ■

The *moveDown* method that takes *element*, the element defining the search path, and moves this search data instance down one level in the tree. It returns the index of the child to which the search data object has moved.

```
protected int moveDown(E element){
    return moveDown(((Node) ptr).childIndex(element, bp));
}
```

Correctness Highlights: This follows from the correctness of *childIndex* and the fact that the *childIndex* method never returns a 1 when called on the sentinel root. The rest of the correctness follows from that of the *moveDown* method that takes the index of the child as its parameter. ■

The *moveUp* method moves the search location to the parent, and returns the branch position for the new search location.

```
protected int moveUp() {
    if (atLeaf()) {                                     //Case 1
        bp = cameFrom.bp();
        ptr = cameFrom;
        cameFrom = (Node) ptr.parent();
        return bp;
    }
    else {                                              //Case 2
        cameFrom = (Node) ptr.parent();
        return super.moveUp();
    }
}
```

Correctness Highlights: By *CAMEFROM*, we have that *cameFrom* correctly holds the previous element in the search path. We consider the two cases.

Case 1: The search location is at a leaf. Since the parent is not an internal node, it is the previous node on the search path. The rest of the correctness follows that of the *bp* and *parent* methods.

Case 2: The search location is not at a leaf. As in Case 1, the parent is the previous node on the search path. The rest of the correctness follows from the *parent* and the inherited *moveUp* method. ■

Recall that the *retraceToLastLeftFork* method takes *x*, the target, and moves the search location to the root of the left fork (if it exists) on the search path defined by *x*. If there is no left fork, then it moves the search location to the root. It requires that this *SearchData* instance is already positioned by a call to *find(x)*. The element *x* need not be an element in the collection.

This method can make use of the knowledge that each Patricia trie node in its role as an internal node has two non-*null* children. Thus, when moving up to find the left fork, if the search location is a left child, it must move up another level. However, when the search location is a right child, its sibling is the left fork root. Note that when the initial search was successful, it is necessary to begin by moving up one level from the leaf to the leaf parent.

```
public void retraceToLastLeftFork(E x) {
    while (!atRoot()) {                                //until search root is reached
        if (atLeaf() || digitizer.numDigits(x) == bp)    //Case 1
            moveUp();
        if (((Node) ptr).childIndex(x, bp) == 0)        //Case 2
            moveUp();
        else {                                           //Case 3
            moveDown(0);
            return;
        }
    }
}
```

Correctness Highlights: Within the loop, the following cases are repeated until either the search position has reached an ancestor for which it was the right child and there is a non-null left child, or until it has been determined that there is no left fork missed since the search position has reached (or began at) the search root.

Case 1: Since no *null* children are encountered, the search path for *x* either ends at a node in its leaf role, or ends at an internal node where the branch position is equal to the number of digits in *x*. In both cases, the search for the last left fork must begin at the parent of the current search location. This case will execute at most once.

Case 2: If the previous *SearchData* location was from the left, then it is necessary to move up the next ancestor to find the first left fork passed.

Case 3: Since every node (except the sentinel root which would only be reached in its leaf role) has two children, if the search location is not a left child, then it must be a right child that has a non-null sibling to the left. Thus the desired search location is obtained by moving to the left child. ■

44.5 Patricia Trie Methods

In this section, we present the methods for the `PatriciaTrie` class.

44.5.1 Constructors

The constructor takes *digitizer*, the digitizer for extracting digits from elements at each branch position. It creates an empty Patricia trie that uses the given digitizer. It throws an *Exception* when the base for the provided digitizer is not 2.

```
public PatriciaTrie(Digitizer<? super E> digitizer) throws Exception {
    super(digitizer);
    if (digitizer.getBase() != 2)
        throw new Exception("Base must be 2");
}
```

The *newNode* factory method takes *element*, the element to place in the new node, and *bp*, the branch position for the new node. It returns a reference to a new `Node` holding the given element and with the given branch position.

```
Node newNode(E element, int bp) {
    if (isEmpty())
        return new Node(-1, element);
    else
        return new Node(bp, element);
}
```

Correctness Highlights: When the collection is empty, the new node becomes the sentinel root, which by definition has a branch position of -1 . Otherwise, the branch position is given by *bp*. ■

The *createSearchData* factory method returns a newly created and initialized `PatriciaSearchData` instance.

```
protected PatriciaSearchData createSearchData() {
    return new PatriciaSearchData();
}
```

Recall that *initSearchData* takes *sd*, the `SearchData` instance to initialize to the search root, and returns the initialized `SearchData` instance.

```
PatriciaSearchData initSearchData(SearchData sd){
    PatriciaSearchData psd = (PatriciaSearchData) sd;
    psd.ptr = root.child(0);
    psd.cameFrom = (Node) root;
    psd.bp = ((CompressedTrieNode<E>) psd.ptr).bp();
    psd.numMatches = 0;
    return psd;
}
```

//reduce casting
//start at search root
//previous node is the sentinel root

Correctness Highlights: By PARENT, the left child of the sentinel root is the search root. CAMEFROM is satisfied by setting *cameFrom* to the sentinel root, which by PARENT is the parent of the search root. By the correctness of *bp()*, BRANCHPOSITION is satisfied. Finally, no matches have yet occurred. ■

44.5.2 Content Mutators

Having each Patricia trie node serve as both an internal and leaf node requires changes in the methods to add and remove elements from the collection.

Methods to Perform Insertion

We first discuss the basic process for inserting a new element *e* into a Patricia trie. As with the compressed trie, first a search is performed for *e*. Let *sd* be the resulting SearchData instance. Insertion in a Patricia trie differs from insertion in a compressed trie in two major ways:

- The new node is always added between *sd.cameFrom* (the predecessor on the search path) and *sd.ptr* (the current search location). That is, the new node is made a child of *sd.cameFrom* and has as children *sd.ptr* (in its current role) and itself (in its role as a leaf). Intuitively, the unsuccessful search for the new node ends at a node whose first distinguishing digit is that of the last branch position tested on the search. Therefore, it is necessary to create a diverging search path at that point.
- Since the branch position stored for every node *x* is computed in terms of *x* in its role as an internal node, *sd.numMatches* is the correct branch position for the new node.

Figure 44.3 illustrates the insertion process. The first element inserted into the Patricia trie is always held in the sentinel root with a left child referencing itself and *null* right child. This situation is handled as a special case.

Next consider the insertion of 1000 into the Patricia trie holding {1111}. After *find* completes, *sd.ptr* references the sentinel root, *sd.cameFrom* also references the sentinel root, and *sd.numMatches* is 1. So the new node (*newNode*) has *newNode.data* = 1000 and *newNode.bp* = 1. Since *sd.ptr* is a left child of *cameFrom*, the new node is added as a left child of *cameFrom*. Finally, the two children of *newNode* will be *sd.ptr* (holding 1111) and itself. By BRANCHPOSITION and PATH, *newNode.bp* identifies the leftmost digit in which *sd.ptr.data()* differs from *newNode.data()*. Whichever of these two elements has a 0 at digit *newNode.bp* becomes the left child and the other (which has a 1 in that digit) becomes the right child. In this example, *newNode* (holding 1000) becomes the left child and *sd.ptr* becomes the right child, yielding the Patricia trie shown at the top right of Figure 44.3.

For ease of exposition we refer to a node holding element *e* as **node** *e*. When node *e* is reached in its leaf role, we refer to it as **leaf** *e*. We now consider the insertion of 001 into the Patricia trie holding {1000, 1111}. After *find* completes, *sd.ptr* is at node 1000 (the search root), *sd.numMatches* is 0 (since comparing 0001 and 1000 from the left results in no matches). Finally, *sd.cameFrom* is node 1111. Thus the new node (001) has a branch position of 0 and is added as a left child of 1111. The new node has itself as its left child and 1000 as its right child. The resulting Patricia trie is shown in the left of the second row of Figure 44.3.

Next consider the insertion of 1001 into the Patricia trie holding {001, 1000, 1111}. After *find* completes, *sd.ptr* is at leaf 1000, *sd.numMatches* is 3 (since comparing 1001 and 1000 from the left results in 3 matches). Finally, *sd.cameFrom* is node 1000. Thus new node (1001) has a branch position of 3 and is added as a left child of 1000 (since 1000 has a branch position of 1 and 1001 has

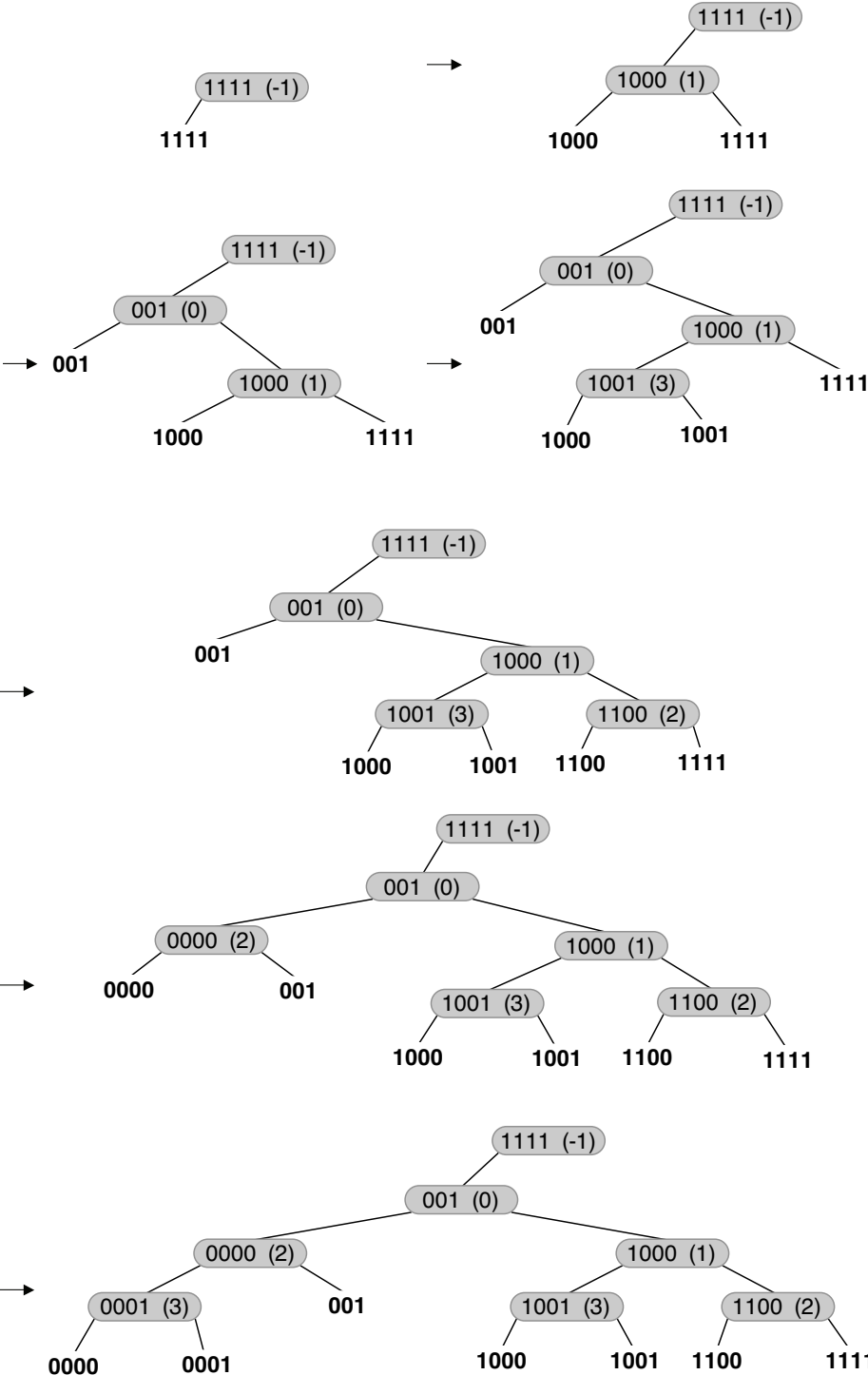


Figure 44.3
Building the Patricia trie of Figure 44.2. Observe that the new node in its leaf role is always a child of the new node in its role as an internal node. That is the new node always has itself as one of its two children.

a 0 at digit 1). The new node has left child of 1000 and a right child of itself. The resulting Patricia trie is shown in the right of the second row of Figure 44.3.

The next insertion is 1100 into the Patricia trie holding {001, 1000, 1001, 1111}. After *find* completes, *sd.ptr* is at leaf 1111, *sd.numMatches* is 2, and *sd.cameFrom* is the node 1000. Thus the new node (1100) has a branch position of 2 and is added as a right child of 1000 (since 1000 has a branch position of 1 and 1100 has a 1 at digit 1). The new node has leaf 1000 as its left child and itself as a right child. The resulting Patricia trie is shown in the right of the third row of Figure 44.3.

The next insertion is 0000 into the Patricia trie holding {001, 1000, 1001, 1100, 1111}. After *find* completes, *sd.ptr* is at leaf 001, *sd.numMatches* is 2, and *sd.cameFrom* is the node 001. Thus the new node (0000) has a branch position of 2 and is added as a left child of 001 (since 001 has a branch position of 0). The new node has itself as its left child and leaf 001 as its right child. The resulting Patricia trie is shown in the right of the fourth row of Figure 44.3.

The final insertion is 0001 into the Patricia trie holding {0000, 001, 1000, 1001, 1100, 1111}. After *find* completes, *sd.ptr* is at leaf 0000, *sd.numMatches* is 3, and *sd.cameFrom* is the node 0000. Thus the new node (0001) has a branch position of 3 and is added as a left child of 0000 (since 0000 has a branch position of 2). The new node has leaf 0000 as its left child and itself as a right child. The resulting Patricia trie is shown in the final row of Figure 44.3.

The *addNewNode* method is very straightforward. In particular, this method takes *newNode*, a trie node that already holds the new element, and *sd*, a search data object positioned by a call to *find* using the element held in *newNode*. This method requires that the new node's branch position has been set to the number of matches that occurred during the search.

```
protected void addNewNode(TrieNode<E> newNode, SearchData sd){
    PatriciaSearchData psd = (PatriciaSearchData) sd;           //to reduce casting
    ((Node) newNode).setChild(newNode);                         //make newNode and sd.ptr
    ((Node) newNode).setChild(sd.ptr);                          //children of newNode
    psd.cameFrom.setChild(newNode);                             //make newNode a child of cameFrom
    psd.ptr = newNode;                                           //reset psd to be at newNode (in leaf role)
    psd.cameFrom = (Node) newNode;                               //so cameFrom is newNode in internal node role
}
```

Correctness Highlights: By the correctness of the *find* method and the requirement placed on *sd*, it follows that *sd* is at the last non-null element in the search path defined by *element*. As discussed, *newNode* should be added as a child of *cameFrom*. Setting its children to itself and *sd.ptr* preserves DESCENDANTOFSELF. The *TrieNode setChild* method maintains PATH, PARENT, and NUMCHILDREN.

Finally, to remove the need for an additional search to update *psd* so that the predecessor can be efficiently found by the *insert* method, we argue that *psd* is correctly updated to hold the values it would have if *find(newNode.data())* were performed after this method had completed. First, since the search would be successful, it would end at *newNode* in its leaf role. So *ptr* must be set to *newNode*. Furthermore, since *newNode* in its leaf role is always a child of *newNode* in its role as an internal node, on the search path to *newNode* in its leaf role, the second to last node on the search path must be *newNode* in its role as an internal node. Thus *cameFrom* must be set to *newNode* (as opposed to its currently value of the parent of *newNode* which is the predecessor of it on the search path in its role as an internal node). ■

Recall that the internal *insert* method takes *element*, the element to be added to the collection. It returns a reference to the new Patricia trie node that was inserted. It throws an *IllegalArgument-Exception* when adding *element* to the collection would violate the requirement that collection is prefix free.


```

protected TrieLeafNode<E> insert(E element) {
    if (isEmpty()) {                                     //special case when collection is empty
        Node newNode = new Node(-1, element);
        root = newNode;
        newNode.children[0] = root;
        newNode.addAfter(FORE);
        size++;
        return newNode;
    }
    PatriciaSearchData sd = (PatriciaSearchData) pool.allocate();
    try {
        FindResult found = find(element, sd);
        if (found == FindResult.PREFIX || found == FindResult.EXTENSION
            || found == FindResult.MATCHED)
            throw new IllegalArgumentException(element + “ violates prefix-free requirement”);
        Node newNode = newNode(element, sd.numMatches);
        addNewNode(newNode, sd);
        found = FindResult.MATCHED;
        if (moveToPred(element, sd, found))
            newNode.addAfter((TrieLeafNode<E>) sd.ptr);
        else
            newNode.addAfter(FORE);
        size++;
        return newNode;
    } finally {
        pool.release(sd);
    }
}

```

Correctness Highlights: The PatriciaTrie *insert* method is almost identical to the Trie *insert* method. There are only two changes. First, because a singleton element is held in the sentinel, a special case is included for inserting an element into an empty collection. It is easily verified that BRANCHPOSITION, PARENT, and DESCENDANTOFSELF are satisfied.

The second change is that the method used to create the new node takes as a parameter *sd.numMatches*. Setting *newNode.bp* to *sd.numMatches* preserves BRANCHPOSITION. The *addNewNode* method preserves PARENT, and DESCENDANTOFSELF.

The rest of the correctness argument is the same as that for the Trie *insert* method. ■

Methods to Perform Deletion

We now discuss the process of removing node *x* from a Patricia trie. Since each Patricia trie node serves as both a leaf and internal node, removing an node requires removing its use in both roles.

Recall that the *removeImpl* method takes *sd*, a search data instance that has been set by a search for the element to be removed. It requires that the structure of the Patricia trie has not changed since *find* was called with *sd.ptr.data*. To avoid repeating the search already performed by *find*, the *removeImpl* method is overridden to call an internal *remove* method that takes a reference to the node to remove and its leaf parent, which is stored in *ptr.cameFrom*.

```

protected void removeImpl(SearchData sd) {
    remove(sd.ptr, ((PatriciaSearchData) sd).cameFrom);
}

```

Correctness Highlights: By the requirement of this method and the correctness of the Patricia search data methods, *sd.cameFrom* is the leaf parent for *sd.ptr*. ■

When the tracker *remove* method is called, only a reference to the Patricia trie node *x* holding the element to remove is known. For node *x*, *x.parent* and *x.bp* hold values with respect to *x* in its role as an internal node, so a reference to *x* does not directly provide the leaf parent of *x*, which is needed to remove *x*. Thus, the *remove* method called used by the tracker, first perform a search from *x* in its role as an internal node to *x* as its role in a leaf node.

The internal *remove* method that takes *x*, a reference to the trie node to remove, searches starting at *x* in its role as an internal node, to find *x* in its role as a leaf node. During the search the leaf parent of *x* is located. Then the *remove* method that takes *x* and its leaf parent can be used to remove *x*.

```

protected void remove(TrieNode<E> x) {
    Node ptrPred = (Node) x;                                //predecessor of ptr on search path
    Node ptr = (Node) ptrPred.child(ptrPred.childIndex(x.data(), ptrPred.bp));
    while (ptrPred.bp < ptr.bp) {                             //until x is reached in leaf role
        ptrPred = ptr;
        ptr = (Node) ptrPred.child(ptrPred.childIndex(x.data(), ptrPred.bp));
    }
    remove(x, ptrPred);
}

```

Correctness Highlights: By BRANCHPOSITION and the correctness of *child*, *ptr* will reach *x* in its leaf role. Since *ptrPred* is always set to the previous value of *ptr*, when the while loop completes, it is the leaf parent for *x*. The rest of the correctness argument follows from the correctness of the *remove* method that takes a reference to both *x* and its leaf parent. ■

We now describe the process used to remove node *x* when also provided with a reference *leafParent* to its leaf parent. One very special case that is easily handled is when there is a single node in the Patricia trie. In this situation the only changes needed are to set *root* to *null* and decrement *size*.

We now consider the situation in which there are at least two elements in the collection. We use *grandparent* to denote the parent of the leaf parent of *x*, and *leafSibling* to denote other child of the leaf parent of *x*. There are four cases that can occur. We first discuss each of these cases in general, and then illustrate them by a sequence of example deletions. In the figures provided to illustrate the general cases, we use the internal representation view as in Figure 44.1 where a dashed arrow highlights child references to a node from its leaf parent. Each triangle represents an arbitrary subtree. Unless an internal node is shown within a triangle, the subtree could be a reference to a single node (in its leaf role). Any parent-child relationships that change when *x* is removed are shown in a thick line in the Patricia trie on the right side of the figure. Finally, for every internal node shown in the illustration (except the sentinel root), the left and right children could be interchanged to yield the symmetric cases.

Case 1: *x* is the sentinel root. This case is illustrated in Figure 44.4. Observe that *leafParent* only has one child other than *x*. Thus, *leafParent* can be removed from its current location

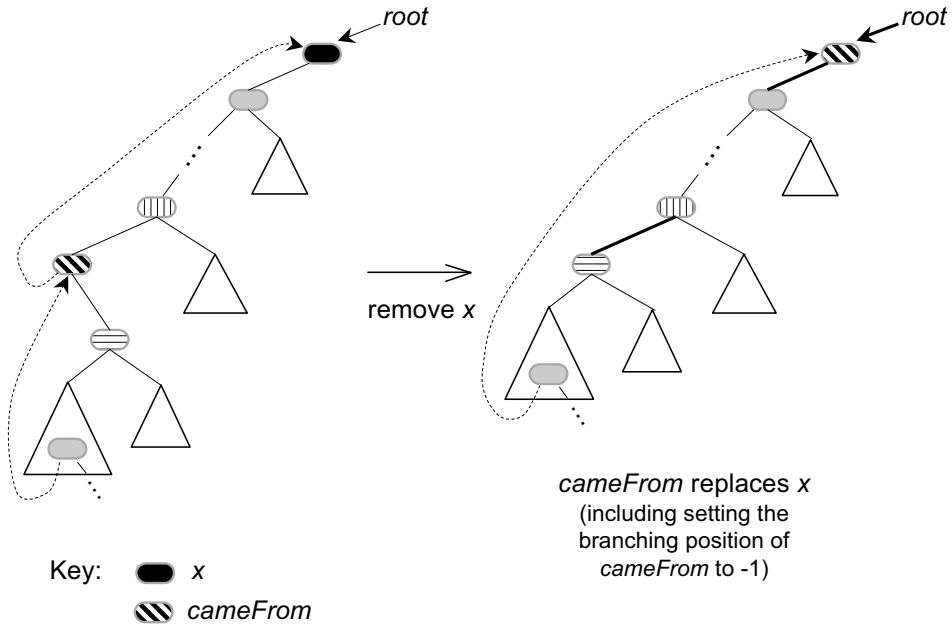


Figure 44.4

Case 1 for deletion from a Patricia trie, when x is the sentinel root.

and be made the new sentinel root. The branch position of *leafParent* is set to -1. However, the branch position does not change for any other node. To remove *leafParent* from its current location, the sibling of x is made a child of *grandparent*. Finally, *root* must be set to *leafParent*.

Case 2: $x \neq \text{root}$ and $x = \text{leafParent}$. In other words, x is its own parent. Figure 44.5 illustrates this case. Observe that x only has one child other than itself. Thus x can be removed by letting x 's parent have as its child *leafSibling*.

Case 3: $x \neq \text{root}$ and $x \neq \text{cameFrom}$. We further divide this into two subcases based on the relationship between x (in its role as an internal node) and *grandparent* (the grandparent of x in its leaf role).

Case 3a: $x = \text{grandparent}$. See Figure 44.6 for a illustration of this case. Since *leafParent* and x both have only one child (except those referencing the other), x can be removed simply by replacing x by *leafParent*. In particular, *leafParent* becomes the child of x 's parent, the branch position of *leafParent* becomes that of x , and finally the other child of x becomes a child of *leafParent*.

Case 3b: $x \neq \text{grandparent}$. Figure 44.7 illustrates this final case, which is the most complex. Here *leafParent* is first removed from its current position by having *grandparent* directly reference the child of *leafParent* that is not x . Then *leafParent* can replace x . More specifically, *leafParent* becomes a child of x 's parent. Then the branch position of x and both of its children replace the branch position and children of *leafParent*.

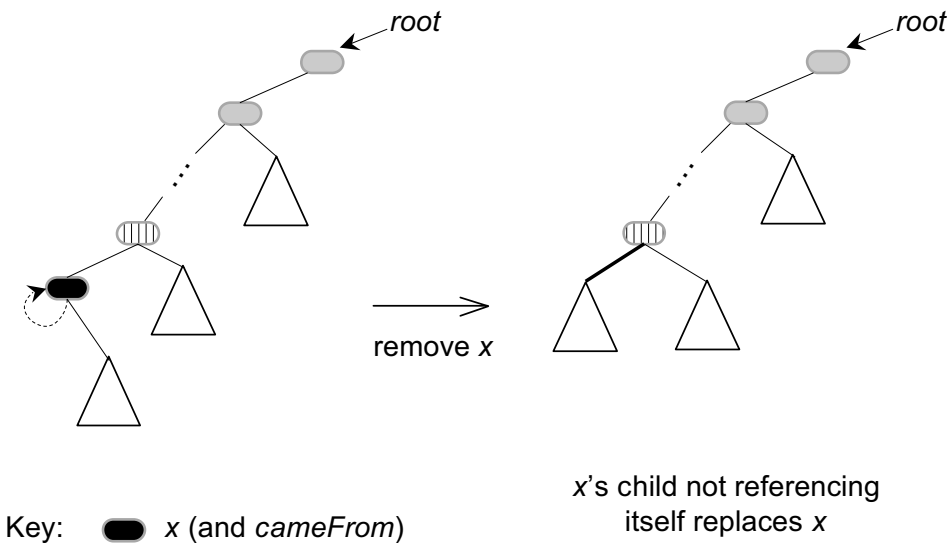


Figure 44.5
Case 2 for deletion from a Patricia trie, when x is its own parent.

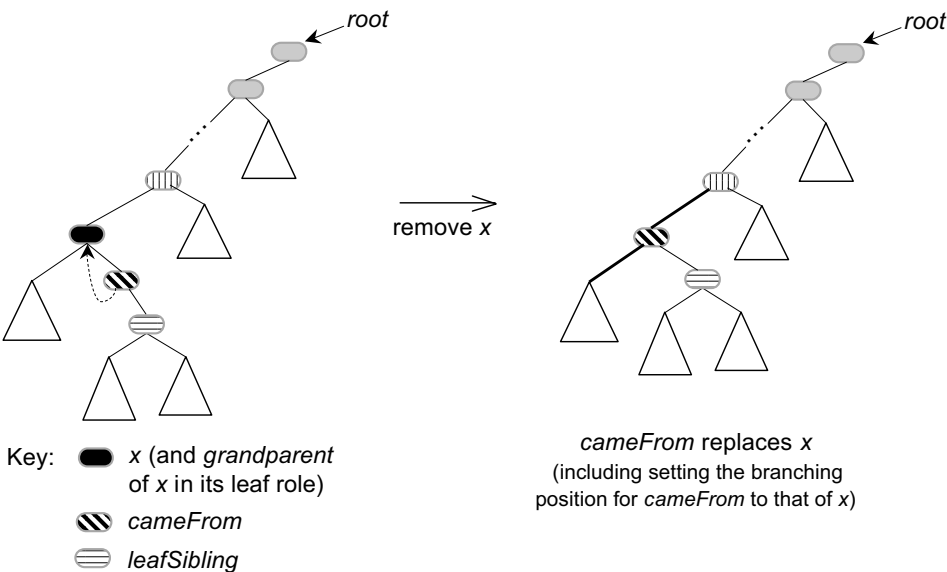


Figure 44.6
Case 3a for deletion from a Patricia trie, when x is its own grandparent.

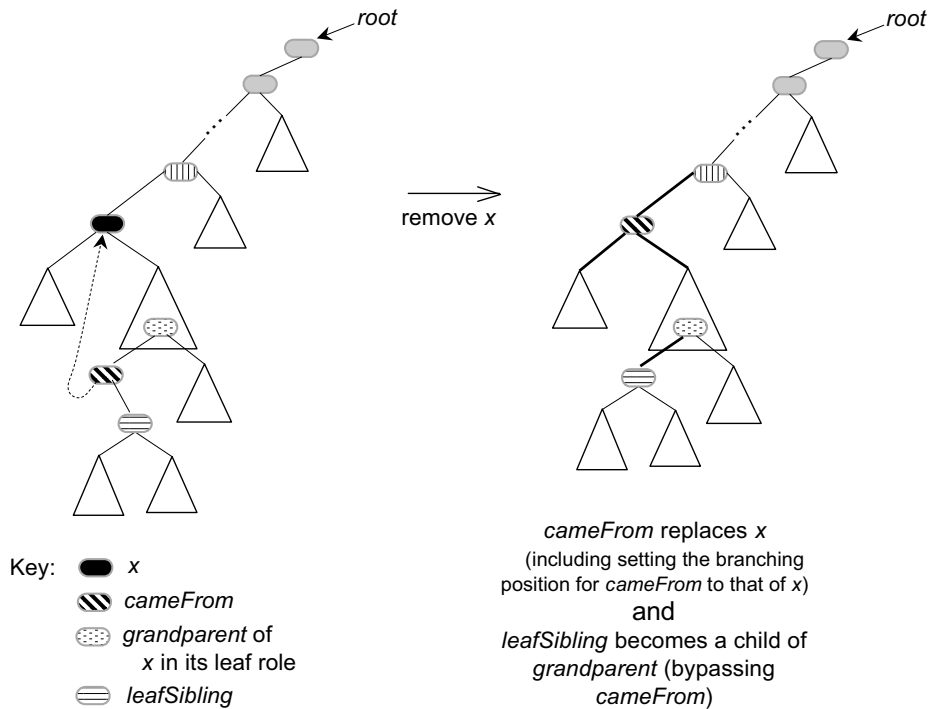


Figure 44.7
Case 3b for deletion from a Patricia trie, when x is not its own grandparent.

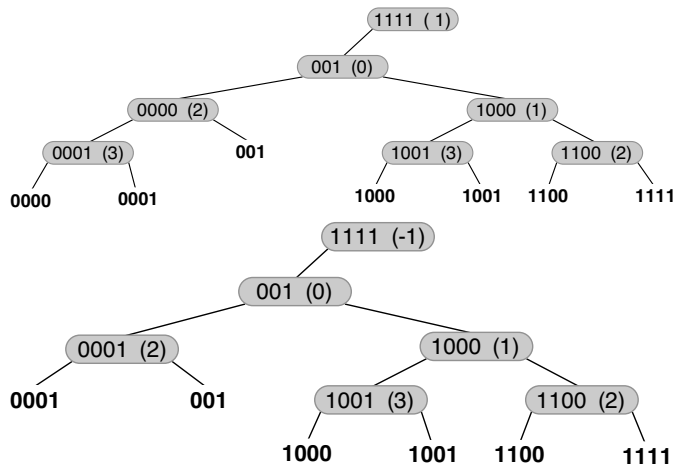


Figure 44.8
Removing 0000 from the Patricia trie shown at the top of the figure is an example of Case 3a, and results in the Patricia trie shown at the bottom of the figure.

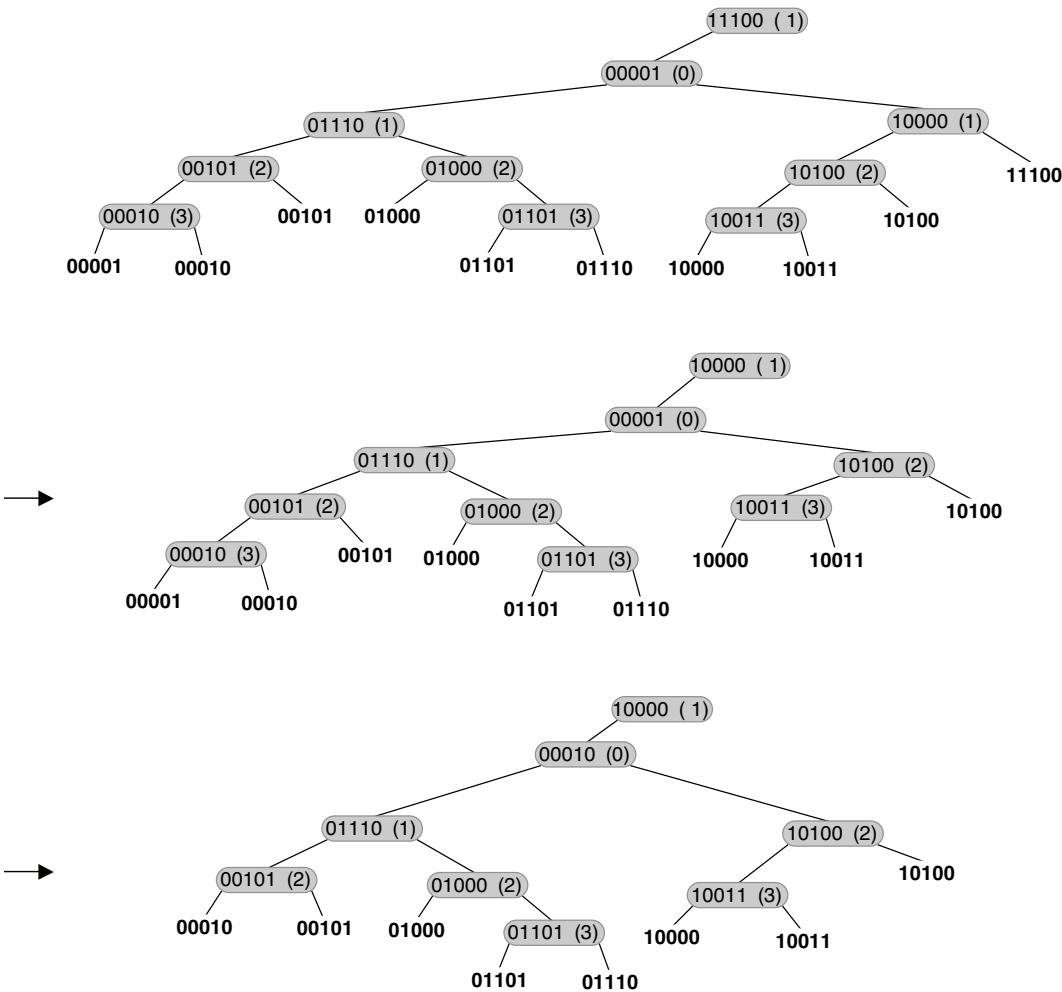


Figure 44.9
Two example deletions in a PatriciaTrie. Removing 11100 illustrates Case 1 and removing 00001 illustrates Case 3b.

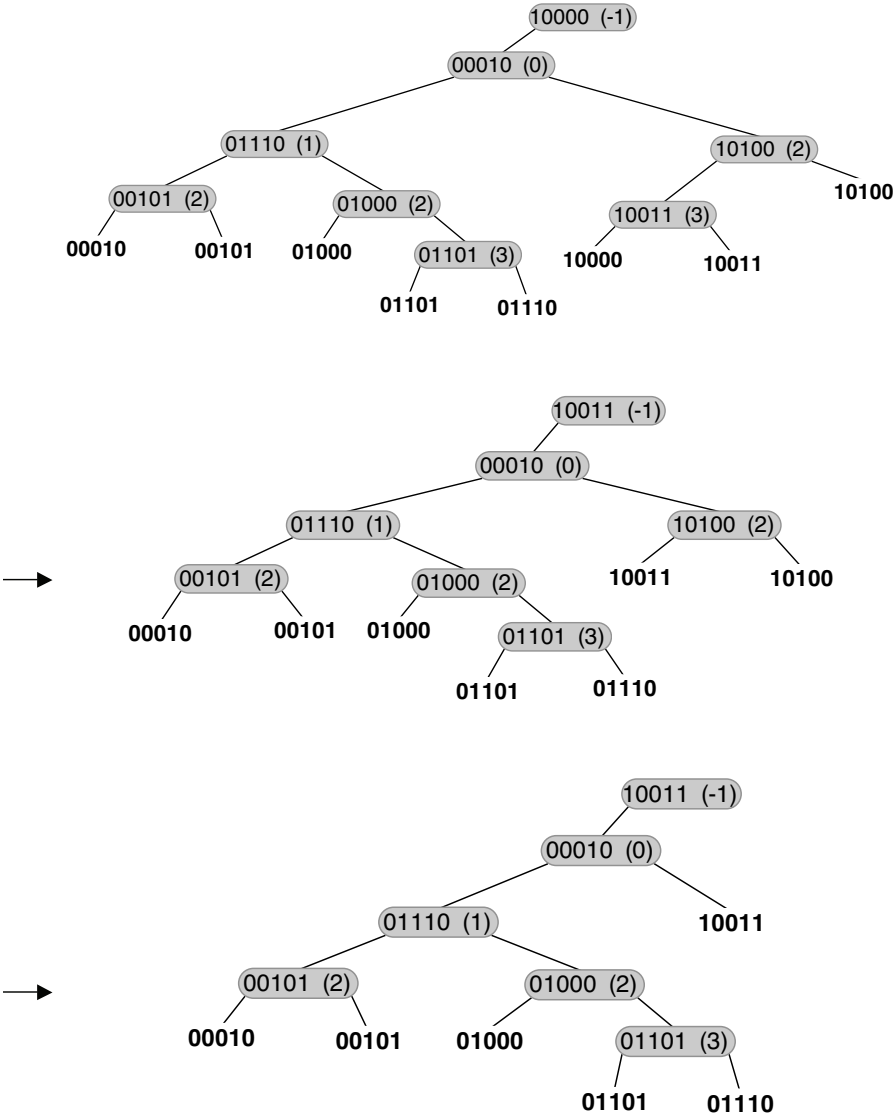


Figure 44.10
Two more example deletions in a PatriciaTrie. Removing 10000 applies Case 1 and removing 10100 illustrates Case 2.

Putting these cases together leads to the following implementation of the *remove* method that takes *x*, a reference to the node to remove, and *leafParent*, a reference to the leaf parent of *x*.

```

protected void remove(TrieNode<E> x, Node leafParent) {
    ((Node) x).remove();                                //preserves OrderedLeafChain
    if (size == 1) {                                     //special case when removing singleton element
        root = null;
        size--;
        return;
    }
    if (x == root) {                                     //Case 1: x is sentinel root
        ((Node) leafParent.parent()).setChild(leafParent.oppositeNode(x.data()));
        leafParent.bp = -1;
        leafParent.setChild(root.child(0));
        leafParent.children[1] = null;
        root = leafParent;
    }
    else {
        Node leafSibling = leafParent.oppositeNode(x.data());    //sibling of x in its leaf role
        Node grandparent = (Node) leafParent.parent();           //grandparent of node in leaf role
        if (leafParent == x)                                       //Case 2a
            grandparent.setChild(leafSibling);                   //make sibling directly child of grandparent
        else {
            leafParent.bp = ((Node) x).bp;                        //Case 3
            ((Node) x.parent()).setChild(leafParent);
            if (grandparent == x) {                                //Case 3a
                leafParent.setChild(((Node) x).oppositeNode(x.data()));
                leafParent.setChild(leafSibling);
            }
            else {                                                //Case 3b
                leafParent.setChild(x.child(0));
                leafParent.setChild(x.child(1));
                grandparent.setChild(leafSibling);
            }
        }
    }
    size--;
}

```

Correctness Highlights: The cases are as described above. By ORDEREDLEAFCHAIN and the correctness of *spliceOut*, INUSE, ORDEREDLEAFCHAIN, and REDIRECTCHAIN are preserved. It is easily verified that if the other properties hold prior to the execution of any of the cases, then the structural properties about the PatriciaTrie are maintained. Since *setChild* is always used to modify any child references, it guarantees that PARENT is preserved. Finally, SIZE is preserved by the decrement of *size*. ■

We close this section by showing some example executions of *remove*. First, as an illustration of Case 3a, 0000 is removed from the Patricia trie shown at the top of Figure 44.8. Here *leafParent* references node 0001, *grandparent* references node 0000 and *leafSibling* is leaf 0001. The result is shown in the bottom of Figure 44.8.

Figure 44.9, illustrates a sequence of two deletions from a different Patricia trie. First 11100 is removed, which applies Case 1 where *leafParent* is node 10000. Next, 00001 is removed. Here *leafParent* is node 00010, *grandparent* is node 00101, and *leafSibling* is leaf 00010. Thus removing 00001 falls under Case 3b. Finally, Figure 44.10, illustrates two more deletions. First 10000 is removed (Case 1) where *leafParent* is node 10011. Next 10100 is removed. Since $x = \text{cameFrom}$ this deletion falls under Case 2.

44.6 Performance Analysis

The Patricia trie is essentially an alternate representation for a compressed trie when $b = 2$. It can take advantage of the fact that all nodes except the sentinel root have two children and that the digitizer cannot be a PrefixFreeDigitizer. However, the asymptotic time complexities for all methods are the same as for the compressed trie.

The major difference between the Patricia trie and compressed trie is the space usage. A Patricia trie of n elements has exactly n Patricia trie nodes, each of which has space allocated for two child references. (With the exception of the sentinel root, all child references are used.) Thus, a Patricia trie has $2n$ child references in the worst case.

44.7 Quick Method Reference

PatriciaTrie Public Methods

- p. 705 *PatriciaTrie*(Digitizer(<? super E> digitizer)
- p. 98 void **accept**(Visitor(<? super E> v)
- p. 659 void **add**(E element)
- p. 100 void **addAll**(Collection(<? extends E> c)
- p. 659 Locator<E> **addTracked**(E element)
- p. 100 void **clear**()
- p. 656 void **completions**(E prefix, Collection(<? super E> c)
- p. 97 boolean **contains**(E value)
- p. 99 void **ensureCapacity**(int capacity)
- p. 652 E **get**(int r)
- p. 96 int **getCapacity**()
- p. 96 Comparator(<? super E> **getComparator**()
- p. 98 E **getEquivalentElement** (E target)
- p. 662 Locator<E> **getLocator**(E x)
- p. 96 int **getSize**()
- p. 96 boolean **isEmpty**()
- p. 661 Locator<E> **iterator**()
- p. 661 Locator<E> **iteratorAtEnd**()
- p. 657 void **longestCommonPrefix**(E prefix, Collection(<? super E> c)
- p. 653 E **max**()
- p. 653 E **min**()
- p. 655 E **predecessor**(E element)
- p. 661 boolean **remove**(E element)
- p. 100 void **retainAll**(Collection<E> c)
- p. 655 E **successor**(E element)
- p. 97 Object[] **toArray**()
- p. 97 E[] **toArray**(E[] array)
- p. 98 String **toString**()
- p. 99 void **trimToSize**()

PatriciaTrie Internal Methods

- p. 657 void **addNewNode**(TrieNode<E> newNode, SearchData sd)
- p. 673 FindResult **checkMatchFromLeaf**(E e, SearchData sd)
- p. 97 int **compare**(E e1, E e2)
- p. 650 SearchData **createSearchData**()
- p. 97 boolean **equivalent**(E e1, E e2)
- p. 651 FindResult **find**(E element, SearchData sd)
- p. 689 FindResult **find**(E target, SearchData sd)
- p. 677 int **getIndexForChild**(InternalNode parent, TrieNode<E> child)
- p. 651 SearchData **initSearchData**(SearchData sd)
- p. 658 TrieLeafNode<E> **insert**(E element)
- p. 656 void **moveToLowestCommonAncestor**(E prefix, SearchData sd, FindResult findStatus)
- p. 653 boolean **moveToPred**(E element, SearchData sd, FindResult findStatus)
- p. 673 TrieNode<E> **newInternalNode**(Object o)
- p. 689 TrieNode<E> **newInternalNode**(int bp, E data)
- p. 651 TrieLeafNode<E> **newLeafNode**(E element)
- p. 705** Node **newNode**(E element, int bp)
- p. 660 void **remove**(TrieNode<E> node)
- p. 716** void **remove**(TrieNode<E> x, Node leafParent)
- p. 661 void **removeImpl**(SearchData sd)
- p. 99 void **traverseForVisitor**(Visitor<? super E> v)
- p. 98 void **writeElements**(StringBuilder s)

PatriciaTrie.Node Public Methods

- p. 638 void **addAfter**(TrieLeafNode<E> ptr)
- p. 687 int **bp**()
- p. 636 TrieNode<E> **child**(int i)
- p. 701** int **childIndex**(E element, int bp)
- p. 636 E **data**()
- p. 638 boolean **isDeleted**()
- p. 636 boolean **isLeaf**()
- p. 638 void **markDeleted**()
- p. 637 TrieLeafNode<E> **next**()
- p. 636 TrieNode<E> **parent**()
- p. 637 TrieLeafNode<E> **prev**()
- p. 638 void **remove**()
- p. 637 void **setNext**(TrieLeafNode<E> nextNode)
- p. 636 void **setParent**(TrieNode<E> parent)
- p. 638 void **setPrev**(TrieLeafNode<E> prevNode)

PatriciaTrie.Node Internal Methods

- p. 700** Node(int bp, E data)
- p. 701** Node **oppositeNode**(E element)
- p. 702** void **setChild**(TrieNode<E> child)
- p. 701** int **setChild**(TrieNode<E> child, E element, int bp)

PatriciaTrie.PatriciaSearchData Public Methods

- p. 702** boolean **atLeaf**()
- p. 702** boolean **atRoot**()
- p. 704** void **retraceToLastLeftFork**(E x)

PatriciaTrie.PatriciaSearchData Internal Methods

- p. 703** int **moveDown**(E element)
- p. 703** int **moveDown**(int childIndex)
- p. 703** int **moveUp**()

Chapter 45

Ternary Search Trie Data Structure

package collection.ordered.digitized

AbstractCollection<E> implements Collection<E>

└ Trie<E> implements DigitizedOrderedCollection<E>, Tracked<E>

└ CompactTrie<E> implements DigitizedOrderedCollection<E>, Tracked<E>

└ TernarySearchTrie<E> implements DigitizedOrderedCollection<E>, Tracked<E>

Uses: Java array and references

Used By: TaggedTernarySearchTrie (Section 49.10.6)

Strengths: Provides the most space efficient DigitizedOrderedCollection implementation when $b > 2$.

Weaknesses: Unlike the trie, compact trie, compressed trie, and Patricia trie in which the worst-case time complexity to find, insert and remove an element is proportional to the number of digits, for a ternary search trie the expected time complexity for these methods is $O(d \log b)$ where d is the number of digits in the given element.

Critical Mutators: none

Competing Data Structures: If the digitizer is base 2 and the collection is naturally prefix free (without using the PrefixFreeDigitizer), then a Patricia trie (Chapter 44) should be used. If $b > 2$ and the time to search for an element is more important than the space usage, then a compressed trie (Chapter 43) should be considered.

45.1 Internal Representation

A ternary search trie is an extension of the compact trie where each internal node just has three children. It can be viewed as a hybrid between a binary search tree (Chapter 32) and a compact trie. As with the compact trie, for an internal node x with a branch position of b , all elements that reach x during a search share a common prefix of b characters. However, the next branch to take depends on both the branching factor and a **comparison element** e associated with the node. Similar to the representation used for the compressed trie, the comparison element is the element held in a leaf element in $T(x)$ that is referenced by the internal node of the ternary search trie.

For comparison element e , let e_b be the b^{th} digit of e . The three children of internal node x are used as follows, where b is the branch position for x and e is the comparison element.

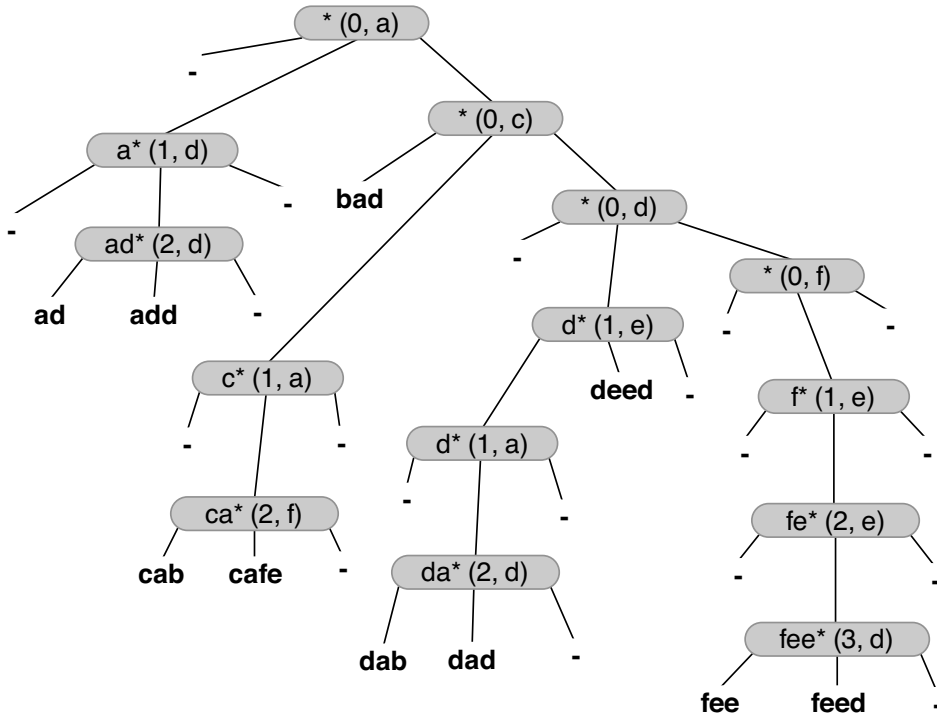


Figure 45.1

A populated example of a ternary search trie holding ad#, bad#, cab#, cafe#, fee#, dab#, feed#, add#, deed#, and dad#. Each internal node x shows $sp(x)$ followed by the branch position, and then the comparison character shown in parentheses.

- All elements in $T(x)$ for which the b^{th} digit is less than e_b are in $T(x.child(0))$.
- All elements in $T(x)$ for which the b^{th} digit is equal to e_b are in $T(x.child(1))$.
- All elements in $T(x)$ for which the b^{th} digit is greater than e_b are in $T(x.child(2))$.

We refer to $child(1)$ as the **descent child** since it is the only one that increases the branch position. The other two children are called the **comparison children**. When a search path uses a comparison child, the branch position does not change, so the same digit must be used again for branching. However, going to a comparison child does reduce the range (number of possible values) for the k^{th} digit where k is the current branch position. In the ideal structure, each step in the path to a comparison child reduces the range by a factor of 2. However, in the worst case, the reduction may be as little as an additive constant of 1. When searching a randomly generated ternary search trie, for each descent child on the path it is expected that $O(\log b)$ steps involving a comparison child will occur. Thus, the expected length of the search path is $(d \log b)$.

Another significant difference from the compact trie is that the structure of a ternary search trie depends on the order in which the elements are inserted. More specifically, what affects the structure is the data associated with each internal node, and the value used for this data depends on the order in which the elements are inserted.

Instance Variables and Constants: All of the instance variables and constants for TernarySearchTrie are inherited from CompactTrie.

Populated Example: Figures 45.1 shows a populated examples of a ternary search trie. Although we illustrate a ternary search trie showing the comparison character for each internal node x , in the internal representation the comparison character is maintained in $x.data$ as a reference to a leaf node in $T(x)$. The comparison character is $digitizer.getDigit(x.data, x.bp)$.

Terminology: We redefine the definition of the search path to reflect the changes that only the descent child changes the branch position, and that the comparison child reduce the range for the digit at the current branch position. We use $start(x)$ and $end(x)$ to denote the start and end of the range for digit $x.bp$.

- We let $d_x = digitizer.getDigit(x.data, x.bp)$.
- We redefine the **search prefix** of internal node x , $sp(x)$ recursively as follows:

$$sp(x) = \begin{cases} \epsilon, & \text{if } x = \text{root} \\ sp(x.parent) + d_x, & x = \text{parent.child}(1) \\ sp(x.parent), & \text{otherwise} \end{cases}$$

where $+$ is the concatenation operator.

- We define $start(x)$ recursively as:

$$start(x) = \begin{cases} 0, & \text{if } x = \text{root} \\ d_{x.parent} + 1, & x = \text{parent.child}(2) \\ start(x.parent), & \text{otherwise.} \end{cases}$$

- We define $end(x)$ recursively as:

$$end(x) = \begin{cases} b, & \text{if } x = \text{root} \\ d_{x.parent} - 1, & x = \text{parent.child}(0) \\ end(x.parent), & \text{otherwise.} \end{cases}$$

Abstraction Function: The ternary search trie abstraction function is the same as that for the trie, compact trie, compressed trie, and Patricia trie. Namely, for ternary search trie T

$$AF(T) = seq(\text{root}).$$

45.2 Representation Properties

The change in the definition of sp requires that both **PATH** and **LEAFPATH** be replaced.

PATH: For reachable trie node x , an element e is in $T(x)$ if and only if $sp(x)$ is a prefix of e and digit $x.bp$ of e is between $start(x)$ and $end(x)$ (inclusive).

LEAFPATH: For every leaf node $\ell \in T(\text{root})$, $\ell.data()$ is an extension of some element in $sp(x)$ for which digit $x.bp$ of $\ell.data()$ is between $start(x)$ and $end(x)$ (inclusive).

45.3 Ternary Search Trie Internal Node Inner Class

`AbstractTrieNode<E>` implements *TrieNode<E>*

└ `TrieNode<E>.InternalNode` implements *TrieNode<E>*

└ **`TernarySearchTrieNode<E>.InternalNode`** implements ***Trie<E>***

The `InternalNode` inner class for `TernarySearchTrie` extends the *Trie<E>.InternalNode* class. It adds an instance variable *dataPtr* that references the leaf node from which the comparison character is drawn. For internal node *x*, the comparison character is digits *x.bp* of *x.dataPtr.data*.

E comparisonElement; //the comparison character element

The constructor takes a single argument *node*, the leaf node holding the comparison character.

```
InternalNode(LeafNode node) {
    comparisonElement = node.data();
}
```

The *data* method returns the data associated with the `LeafNode dataPtr`.

```
public E data() {
    return comparisonElement;
}
```

Recall that the *childIndex* method takes *element*, the element with respect to the index of the appropriate child is sought, and *bp*, the branch position of this node. It returns the index of the child followed by *sp(element)*, assuming the current node has branch position *bp*. We override this method here to determine which branch to take 0 (left), 1 (middle) or 2 (right) based on the relation between the comparison character and *bp* digit of the element.

```
public int childIndex(E element, int bp){
    int elementDigit = digitizer.getDigit(element, bp);
    int dataDigit = digitizer.getDigit(data(), bp);
    if (elementDigit < dataDigit)
        return 0;
    else if (elementDigit == dataDigit)
        return 1;
    else
        return 2;
}
```

45.4 Ternary Search Trie Search Data Inner Class

SearchData

└ ***TernarySearchTrieSearchData***

We override the `SearchData` inner class of the `Trie` class since the branch position changes only when moving down to or coming up from the middle child.

Recall the `childBranchPosition` method takes `childIndex`, the index for the child of interest. It returns the branch position of that child.

```
int childBranchPosition(int childIndex){
    return (childIndex == 1) ? bp+1 : bp;
}
```

Correctness Highlights: The branch position is properly incremented only for the descent child. ■

The `parentBranchPosition` method returns the branch position of the parent.

```
int parentBranchPosition(){
    return (ptr.parent().child(1) == ptr) ? bp-1 : bp;
}
```

Correctness Highlights: Since the branch position is incremented only for the descent child, it correctly only decrements when moving up from the descent child. ■

Recall that the `processedEndOfString` method takes `element`, the element defining the search path. It returns `true` if and only if the last step of the search path occurred by processing the end of string character.

```
protected boolean processedEndOfString(E element) {
    return (digitizer.isPrefixFree() && ptr != root &&
        parentBranchPosition() == digitizer.numDigits(element)-1);
}
```

Correctness Highlights: First, the end of string character can only be processed if the digitizer is a prefix free digitizer. Second, the end of string character cannot be processed if the search location is at the root since in that case no digits have been processed. By `BRANCHPOSITION`, the branch position for a node is its level in the ternary search trie. Thus the end of string character has been processed exactly when the parent branch position is equal to the number of digits in the element (excluding the end of string character). ■

45.5 Ternary Search Trie Methods

In this section we describe the methods of `CompactTrie` that we override for `TernarySearchTrie`.

45.5.1 Constructors and Factory Methods

The constructor takes `digitizer`, the digitizer to extract digits from elements. It creates an empty ternary search trie that uses the given digitizer.

```

public TernarySearchTrie(Digitizer<? super E> digitizer) {
    super(digitizer);
    childCapacity = 3;           //only three children per node, regardless of the base
}

```

The *createSearchData* factory method returns a newly created and initialized ternary search trie search data instance.

```

protected SearchData createSearchData() {
    return initSearchData(new TernarySearchTrieSearchData());
}

```

The *newLeafNode* factory method takes *element*, the data element, and returns a newly created ternary search trie leaf node holding the given element.

```

protected TrieLeafNode<E> newLeafNode(E element, int level) {
    return new LeafNode(element);
}

```

The *newInternalNode* factory method takes *o*, an object to initialize the data value for the new node. It creates and returns a new ternary search trie node holding *o*.

```

protected TrieNode<E> newInternalNode(Object o) {
    return new InternalNode((LeafNode) o);
}

```

45.5.2 Algorithmic Accessors

There is only one significant change required for the TernarySearchTrie (other than those made in the internal node and search data inner classes). Recall that the internal *moveToLowestMatchingAncestor* method takes *prefix*, a digitized element, *sd*, a SearchData instance that has been set by a search for *prefix*, and *found*, the FindResult value returned by the search that was used to set *sd*. This method has the side affect of moving *sd* so that it is at the lowest common ancestor for which the associated data is an extension of *prefix*.

```

protected void moveToLowestCommonAncestor(E prefix,
                                           SearchData sd, FindResult found) {
    while (!(sd.ptr == root || sd.ptr == sd.ptr.parent().child(1) ||
           (found == FindResult.MATCHED && !sd.processedEndOfString(prefix))))
        sd.moveUp();
}

```

Correctness Highlights: Recall that *moveToLowestMatchingAncestor* is used by *longestMatchingPrefix(element)*. It should move to the parent in the search path as long as a mismatch has occurred. Also, if the end of string character was processed then it is important to move to the parent since there could be extensions that occur with *element* as a prefix. There are three conditions that define when *sd* should no longer be moved up.

- When the root is reached, it is the lowest common ancestor.

- If *sd.ptr* is a middle child, then it follows that *sp(sd.ptr)* is a prefix of *element*. Observe that the search (using *find*) that positions *sd* follows a sequence of matching characters stopping at the first mismatched character. Thus, when retreating toward the root, once a matching character is reached, all earlier characters also matched. So *sp(sd.ptr)* is a longest common prefix of *element*.
- If a match occurred, *sd* must be moved up unless the end of string was processed. In other words, the loop should no longer execute if there was a match and the end of string character was not processed. (Really, the end of string was processed, but once we have moved up, it is “as if” it was not processed, so *sd* returns the correct value, *false*, the next time through the loop.)

45.6 Performance Analysis

The asymptotic time complexities of all public methods for the Trie class are shown in Table 45.2, and the asymptotic time complexities for all of the public methods of the TernarySearchTrie Tracker class are given in Table 45.3.

A ternary search trie can be thought of as a hybrid between a trie and a binary search tree. The descent child has the behavior of a trie and the comparison children have the behavior of a binary search tree. Intuitively, after going to a descent child, the comparison children form a binary search tree that is used to find the next internal node for which the descent child can be taken. Thus the expected height of a ternary search trie (equivalently, the length of the search path) is larger by a multiplicative factor of $\log_2 b$.

Thus, a method with $O(h)$ complexity for a trie will have a worst-case time complexity of $O(b \cdot h)$ for a ternary search trie, and an expected time complexity of $O(\log b \cdot h)$ for a ternary search trie. Recall that the expected path length in a trie is $O(\log n / \log b)$. So the expected height for a ternary search trie constructed from random elements is $O(\log n)$, which is independent of b .

Clément et al. [40] prove that for a ternary search trie constructed from randomly generated elements, the expected number of child references is $3(s - 1 + n)$, where s is the number of internal nodes in the compact trie holding the same elements. Thus, the expected number of references in a ternary search trie is roughly $3n \left(1 + \frac{1}{\ln b}\right)$.

45.7 Quick Method Reference

TernarySearchTrie Public Methods

- p. 723 TernarySearchTrie(Digitizer(? super E) digitizer)
- p. 98 void **accept**(Visitor(? super E) v)
- p. 659 void **add**(E element)
- p. 100 void **addAll**(Collection(? extends E) c)
- p. 659 Locator(E) **addTracked**(E element)
- p. 100 void **clear**()
- p. 656 void **completions**(E prefix, Collection(? super E) c)
- p. 97 boolean **contains**(E value)

method	worst-case time complexity	expected time complexity
constructor	$O(1)$	$O(1)$
<i>ensureCapacity(x)</i>	$O(1)$	$O(1)$
<i>iterator()</i>	$O(1)$	$O(1)$
<i>iteratorAtEnd()</i>	$O(1)$	$O(1)$
<i>trimToSize()</i>	$O(1)$	$O(1)$
<i>add(o')</i>	$O(b \cdot \min(d, d_{\max}))$	$O(\log n)$
<i>addTracked(o')</i>	$O(b \cdot \min(d, d_{\max}))$	$O(\log n)$
<i>contains(o')</i> , unsuccessful	$O(b \cdot \min(d, d_{\max}))$	$O(\log n)$
<i>min()</i>	$O(b \cdot d_{\max})$	$O(\log n)$
<i>max()</i>	$O(b \cdot d_{\max})$	$O(\log n)$
<i>predecessor(o')</i>	$O(b \cdot d_{\max})$	$O(\log n)$
<i>successor(o')</i>	$O(b \cdot d_{\max})$	$O(\log n)$
<i>contains(o)</i> , successful	$O(db)$	$O(d \log b)$
<i>getEquivalentElement(o)</i>	$O(db)$	$O(d \log b)$
<i>getLocator(o)</i>	$O(db)$	$O(d \log b)$
<i>remove(o)</i>	$O(db)$	$O(d \log b)$
<i>completions(o')</i>	$O(b \cdot d_{\max} + s)$	$O(\log n + s)$
<i>longestCommonPrefix(o')</i>	$O(b \cdot d_{\max} + s)$	$O(\log n + s)$
<i>clear()</i>	$O(n)$	$O(n)$
<i>toArray()</i>	$O(n)$	$O(n)$
<i>toString()</i>	$O(n)$	$O(n)$
<i>accept()</i>	$O(n)$	$O(n)$
<i>addAll(c)</i>	$O(b \cdot c \cdot d_{\max})$	$O(c \cdot \log(n + c))$
<i>retainAll(c)</i>	$O(b \cdot n(c + d_{\max}))$	$O(n(c + \log n))$

Table 45.2 Summary of the asymptotic time complexities for the `DigitizedOrderedCollection` public methods when using the ternary search trie data structure. We use the convention that o is in the collection, and o' is not in the collection. The expected time complexities were derived under the assumption that all elements in the trie and o and o' , are random elements, where the number of digits approaches infinity. The right column shows the expected behavior as a function of n when the number of digits is $\omega(\log_b n)$. We use d to denote the number of digits in the object o (or o'), $d_{\max}$ to denote the maximum number of digits in any element in the collection, and s to denote the number of elements in a positional collection returned.

locator method	worst-case time complexity	expected time complexity
constructor	$O(1)$	$O(1)$
<i>advance()</i>	$O(1)$	$O(1)$
<i>get()</i>	$O(1)$	$O(1)$
<i>hasNext()</i>	$O(1)$	$O(1)$
<i>next()</i>	$O(1)$	$O(1)$
<i>retreat()</i>	$O(1)$	$O(1)$
<i>remove()</i>	$O(db)$	$O(d \log b)$

Table 45.3 Summary of the time complexities for the public Locator methods of the Trie Locator class. We use b to denote the base for the digitized elements and d to denote the number of digits in the element to be removed. Any methods that take constant time for all OrderedCollection Locators are not shown.

- p. 99 void **ensureCapacity**(int capacity)
- p. 652 E **get**(int r)
- p. 96 int **getCapacity**()
- p. 96 Comparator<? super E> **getComparator**()
- p. 98 E **getEquivalentElement** (E target)
- p. 662 Locator<E> **getLocator**(E x)
- p. 96 int **getSize**()
- p. 96 boolean **isEmpty**()
- p. 661 Locator<E> **iterator**()
- p. 661 Locator<E> **iteratorAtEnd**()
- p. 657 void **longestCommonPrefix**(E prefix, Collection<? super E> c)
- p. 653 E **max**()
- p. 653 E **min**()
- p. 655 E **predecessor**(E element)
- p. 661 boolean **remove**(E element)
- p. 100 void **retainAll**(Collection<E> c)
- p. 655 E **successor**(E element)
- p. 97 Object[] **toArray**()
- p. 97 E[] **toArray**(E[] array)
- p. 98 String **toString**()
- p. 99 void **trimToSize**()

TernarySearchTrie Internal Methods

- p. 657 void **addNewNode**(TrieNode<E> newNode, SearchData sd)
- p. 673 FindResult **checkMatchFromLeaf**(E e, SearchData sd)
- p. 97 int **compare**(E e1, E e2)
- p. 650 SearchData **createSearchData**()
- p. 97 boolean **equivalent**(E e1, E e2)
- p. 651 FindResult **find**(E element, SearchData sd)
- p. 677 int **getIndexForChild**(InternalNode parent, TrieNode<E> child)
- p. 651 SearchData **initSearchData**(SearchData sd)
- p. 658 TrieLeafNode<E> **insert**(E element)
- p. 724 void **moveToLowestCommonAncestor**(E prefix, SearchData sd, FindResult found)
- p. 656 void **moveToLowestCommonAncestor**(E prefix, SearchData sd, FindResult findStatus)
- p. 653 boolean **moveToPred**(E element, SearchData sd, FindResult findStatus)
- p. 673 TrieNode<E> **newInternalNode**(Object o)
- p. 651 TrieLeafNode<E> **newLeafNode**(E element)

p. 724 `TrieLeafNode<E> newLeafNode(E element, int level)`
p. 660 `void remove(TrieNode<E> node)`
p. 661 `void removeImpl(SearchData sd)`
p. 99 `void traverseForVisitor(Visitor<? super E> v)`
p. 98 `void writeElements(StringBuilder s)`

TernarySearchTrie.InternalNode Public Methods

p. 636 `TrieNode<E> child(int i)`
p. 643 `int childIndex(E element, int bp)`
p. 636 `E data()`
p. 636 `boolean isLeaf()`
p. 636 `TrieNode<E> parent()`
p. 636 `void setParent(TrieNode<E> parent)`

TernarySearchTrie.InternalNode Internal Methods

p. 722 `InternalNode(LeafNode node)`
p. 643 `int setChild(TrieNode<E> child, E element, int bp)`

TernarySearchTrie.TernarySearchTrieSearchData Internal Methods

p. 723 `int childBranchPosition(int childIndex)`
p. 723 `int parentBranchPosition()`
p. 723 `boolean processedEndOfString(E element)`

Chapter 46

Spatial Collection ADT

package collection.spatial

Collection<*E*>

↳ *SpatialCollection*<*E*>

A spatial collection organizes its elements by location in a multidimensional space. Example physical data sets include landmarks on a 2-dimensional street map, and star and planet coordinates in space. Data sets of higher dimension might include meteorological data with dimensions in space, time, barometric pressure, temperature, wind velocity, etc. Since each object can be treated as a point in a multidimensional space, we often refer to each object as a **point**.

Spatial collection implementations, including the k -d tree (Chapter 47) and the quad tree (Chapter 48), typically support methods that concern location and order in a k -dimensional space, such as finding the element with the minimum value for a specified dimension, or finding which elements fall within a given multidimensional bounding box, such as finding all restaurants within the four square blocks around a given street corner. Observe that a point $p = (p_1, \dots, p_k)$ is in the box defined by the minimum corner $(a_1, \dots, a_k)$ and the maximum corner $(b_1, \dots, b_k)$ exactly when for all $1 \leq i \leq k$, $a_i \leq p_i \leq b_i$. This operation is known as an **orthogonal range search** (or **orthogonal range query**). Observe that an orthogonal range query generalizes the 1-dimensional range search. For an example application of a spatial collection, see the collision detection case study of Section 46.1.

Most *SpatialCollection* ADT implementations extend an ordered collection data structure (which can be thought of as 1-dimensional spatial collection) in some way. For example, a k -d tree (Chapter 47) builds upon a binary search tree by cycling through the k dimensions in deciding whether an element is placed in the left or right subtree. A quad-tree (Chapter 48 or its higher dimensional variants, instead has 2^k branches at each level, one corresponding to each possible relationship between the target element and the element associated with the node. As another example, an R -tree discussed briefly in Section 46.5 builds upon a B -tree (Chapter 36).

46.1 Case Study: Collision Detection in Video Games

In this section, we describe two of the many ways in which spatial collections are used in video game development. Consider a simple maze game in which a virtual 2-dimensional environment is composed of rectilinear obstacles through which that the player must navigate to reach a goal. If a collision occurs with one of the obstacles then the player is sent back to the start, perhaps with some penalty, to try again. The player's location can be viewed as a point in the 2-dimensional domain. Many times a second, the following task must be performed. Given the player's current location p , determine if the player is at one of the boundaries of an obstacle, or inside the obstacle. If so, a collision has occurred. Observe that while p might be just inside an obstacle, since you do not check often enough to notice the exact moment when p reaches the boundary, p never appears to be inside the object.

For this application, a spatial collection, particularly a k -d tree or quad tree, can be used as a way

to partition the domain into rectangles (or boxes in higher dimensions) such that every rectangle is either entirely outside an obstacle, or entirely on or inside the boundaries of an obstacle. For any point p , a collision has occurred if and only if p is in a region that is on or inside an obstacle boundary. Let T be the k -d tree (or quad tree) used to maintain the obstacles in player's environment. To partition the region into rectangles, every corner of the obstacle is inserted into T . This choice implies that each partition of the domain is based on an axis-aligned plane that goes along a side of one of the obstacles. Thus every leaf corresponds to a region of the domain that is not intersected by any boundary of an obstacle. This implies the desired property that the region corresponding to each leaf is entirely inside (or on) an obstacle, or entirely outside the obstacle. Finally, given a user position p , a search for the leaf in T corresponding to the user position can be performed in logarithmic time. Often this application of a spatial collection is known as **spatial hashing** since it can be viewed as a hash table designed for a point in a d -dimensional space.

A point quad tree (Chapter 48) partitions the space according to the four quadrants defined by an element (i.e., a point) in the collection. At each internal node of T , a search for p requires determining which quadrant contains p . Thus, like a binary search tree (Chapter 32) both the internal and leaf nodes of a point quad tree are associated with a point in the collection. A **point region (PR) quad tree** is a variation of a quad tree in which all elements in the collection are stored in leaf nodes (as in a B+-tree discussed in Chapter 37). For the application being considered in this case study, the domain is typically a 2^x by 2^x region (e.g., 1024 by 1024). To both minimize space usage and improve the search time for this application, the following variation of a point region (PR) quad tree is often used. The root of the quad tree splits the domain into four equal-sized regions (independently of the points in the collection). Then for each subregion in which the number of elements is greater than some preselected threshold, this subdivision process is recursively repeated. The advantage of this data structure is that by examining only 1 bit for each dimension, the appropriate quadrant can be found. Furthermore, the only information that must be stored for each internal node is the array of references to its four children. At the root, the most significant bit in each dimension defines which branch to take, at the next level the second most significant bit can be used, and so on. The drawback of this approach is that the only aspect of the resulting partition that depends on the specific collection being represented is when to recursively partition a node. However, if the obstacles are fairly evenly spread throughout the domain, this approach can work well in practice. Generally, obstacles cannot be defined using axis-parallel boundaries. When the environment is static, a BSP tree [68] is often used since it does not require the obstacles to be rectilinear.

We close by briefly exploring one other task. Suppose the game developer wanted to have any obstacle within a given vertical distance and given horizontal distance from the player perform some action (e.g., project an object towards the player). That is, for player position $p = (p_x, p_y)$, with horizontal tolerance of Δ_x , and vertical tolerance of Δ_y , the task is to find all obstacles that contain some point $p' = (p'_x, p'_y)$ where $p_x - \Delta_x \leq p'_x \leq p_x + \Delta_x$ and $p_y - \Delta_y \leq p'_y \leq p_y + \Delta_y$. By appropriately sampling points from the perimeter of each obstacle, an orthogonal range query with corners $(p_x - \Delta_x, p_y - \Delta_y)$ and $(p_x + \Delta_x, p_y + \Delta_y)$ provides a good approximation. If the objects are complex, too many sample points may need to be placed into the spatial collection. In such cases the smallest bounding box can be used for an initial approximation, and then a more refined search can be performed with any obstacle for which its bounding box is returned by the orthogonal range query.

46.2 Interface

A spatial collection has the following methods, in addition to those inherited from the *Collection* interface.

SpatialCollection(Comparator<? super E> ... comparators): Creates a new empty spatial collection, where the provided comparators define the dimensions along which data are compared. Each dimension is assigned an index (0, 1, ...) that is fixed according to the order in which the comparators are provided as parameters to the constructor.

E max(int dimension): Returns a greatest element in the collection along the given dimension. This method throws a *NoSuchElementException* when the collection is empty. It throws an *IllegalArgumentException* when the given dimension index is not valid for this spatial collection.

E min(int dimension): Returns a least element in the collection along the given dimension. This method throws a *NoSuchElementException* when the collection is empty. It throws an *IllegalArgumentException* when the given dimension index is not valid for this spatial collection.

Collection<E> withinBounds(E minCorner, E maxCorner): Returns a collection of the elements that fall within (or on) the boundary of the multidimensional box defined by the two given corners, *minCorner* and *maxCorner*. That is, this method performs an orthogonal range search. It requires that the coordinates of *minCorner* are less than or equal to those of *maxCorner* along every dimension of the spatial collection.

Critical Mutators: *remove*

46.3 Competing ADTs

We briefly discuss ADTs that may be appropriate in a situation when a *SpatialCollection* ADT is being considered.

TaggedSpatialCollection ADT: If it is more natural for the application to associate a tag, which is a k -dimensional point (for $k \geq 2$), and it is uncommon for multiple elements to have the same tag, then a tagged spatial collection is more appropriate. For example suppose an application needs to maintain a set of readings (with longitude, latitude and time) for a set of mobile devices. An operation might be needed to efficiently return a collection of all devices that were in a certain geographic region during a given time range. This data could be maintained by associating a three-dimensional tag with the mobile device from which the reading was taken.

TaggedBucketSpatialCollection ADT: If it is more natural for the application to associate a tag, which is a k -dimensional point, and it is common for multiple elements to have the same tag, then a tagged bucket spatial collection is more appropriate. Suppose, in the example above, that the time stamp only includes the day, and it is expected that there will be many readings taken on a given day from a given location. For such an application a grouped tagged spatial collection is good choice since it supports efficient retrieval of the set of all devices that were near a certain location during a specified time period.

OrderedCollection ADT: If there is a single ordering for the elements (i.e., a one-dimensional space) then an ordered collection is a better choice. In particular, a skip list (Chapter 38) provides very good support for a one-dimensional range search.

46.4 Summary of Spatial Collection Data Structures

KDTree: The k -d tree cycles among the k dimensions, dividing each region by a halfspace with respect to the dimensions associated with that level. Since the partition at each node uses just one dimension, a k -d tree has the flexibility needed to efficiently partition the domain for certain data distributions (e.g., points along the diagonal). Because of this flexibility, the space usage of a k -d tree scales well in the number of dimensions, and so it is the spatial collection data structure of choice for $k > 3$. While the expected height of a k -d tree is roughly k times more than a k -dimensional extension of a quad tree, each level only requires a single comparison in a k -d tree, versus k comparisons in a k -dimensional extension of a quad tree. Also, deletion is much less complex in a k -d tree than it is for a quad tree.

QuadTree: A quad tree divides the subdomain into four regions at each internal node. In general, if there are k dimensions, one can think of each node (in the k -dimensional extension of the quad tree) as grouping k consecutive levels of a k -d tree, giving a combined branching factor of 2^k . At each node, a comparison is made along all k dimensions. To do this compression, a single k -dimensional point is used to partition the domain into 2^k subdomains. Thus, the expected height of a k -dimensional extension of quad tree is less than the expected height of a k -d tree by a factor of k . While the expected number of comparisons is the same, the larger expected depth in a k -d tree slightly increases the search time. Furthermore, if there is support to make up to 3 comparisons in parallel, then a quad tree (for $k = 2$) or an oct tree (for $k = 3$) leads to significantly faster expected search times since all k comparisons can be made in parallel whereas for a k -d tree the comparisons must be made sequentially. A k -d tree could also perform two levels of comparison in parallel, but an extra comparison would then be made.

46.5 Further Reading

Discussion of spatial collections can be found in books by Goodrich and Tamassia [76], Melhorn [113], Samet [131, 132], Wood [158]. Much research has focused on data structures for static spatial collections, in which all elements are known when the data structure is constructed. We focus here on data structures for a dynamic spatial collection, in which elements can be added and removed.

The k -d tree was proposed by Bentley [21]. The **adaptive k -d tree** is a variation of the k -d tree in which the comparator used does not rotate between the levels, but instead the dimension that divides the remaining elements most evenly is selected as the discriminator.

Quad trees were first presented by Finkel and Bentley [53]. Samet [130] described the more efficient deletion procedure that is used by our implementation. These quad trees are known as a **point quad tree** since each internal node is associated with an element (point) in the collection. A **point region (PR) quad tree** is a variation in which the elements are only held at the leaves. Each internal node corresponds to selecting a point in the domain that separates the elements in the subregion as evenly as possible among its four children. The **balanced box decomposition tree** is a hybrid between a quad tree and a k -d tree [14, 15].

Guttman [81] proposed **R-trees**, which are based on B-trees (Chapter 36). An R-tree splits the domain with hierarchically nested, and possibly overlapping bounding boxes. As in a B-tree, the number of children for each node can vary. So like B-trees, R-trees are a good choice when secondary storage is required since they can minimize the number of disk accesses required to locate

a desired element, or to perform an orthogonal range search. Two variations of the R-tree are the **quadratic R-tree** and **linear R-tree**, which differ only in the algorithm used to split a full node. Although R-trees do not guarantee good worst-case performance, they perform well in practice. The **priority R-tree** is a variation of an R-tree that provides worst-case guarantees on performance [13]. There are other variations including the R+-tree [138] and R*-tree [19].

The k -d tree, quad tree, and R-tree fall into the broad class of **partition based** spatial collection data structures that partition the domain into disjoint regions using axis-aligned halfspaces. There are some variations that use non-rectilinear partitioning components. For example, a **binary space partition tree** (BSP tree) is a variation of a k -d tree in which the partition of a region need not be axis parallel [68], and a **hierarchical-cutting tree** can answer triangular range queries [107, 108].

There are also many spatial collection data structures that do not partition the domain, but rather are more direct extensions of standard (1-dimensional) search trees. For example, a **two-dimensional range tree** defines one comparator as the **primary comparator**, and the other comparator becomes the **secondary comparator**. The primary data structure is a balanced search tree based upon the primary comparator. Each node v in the search tree is associated with a secondary balanced search tree that holds all elements in $T(v)$ organized according to the secondary comparator.

The **Cartesian tree** described by Jean Vuillemin [154] in 1980 and the **priority search tree** described by McCreight [109] in 1985 create a binary search tree (or balanced search tree) based on the primary comparator, and then use the secondary comparator to assign a priority to each node. Rotations are used to maintain the property that the priority of each node is never larger than that of its parent. (The **treap** of Seidel and Aragon [137] is essentially the same data structure, except that they randomly assign the priority as an alternate way to balance a search tree. When the priority is selected at random, it is often called a **randomized binary search tree**.) An advantage of this data structure over the two-dimensional range tree is that it only requires linear space. McCreight [109] showed how a priority search tree could be used to answer a restricted form of a range search, in which no upperbound is given for the secondary comparator. Edelsbrunner [49] showed how priority search trees can be used to answer two-dimensional orthogonal range queries.

Chapter 47

KD-Tree Data Structure

package `collection.spatial`

`AbstractCollection<E>` implements *Collection<E>*

↳ `KDTree<E>` implements *SpatialCollection<E>*

Uses: `DoublyLinkedList` (Chapter 16), `BinarySearchTree` (Chapter 32)

Used By: `TaggedKDTree` (Section 49.11.2)

Strengths: The space usage of a k -d tree scales well in the number of dimensions, so it is the spatial collection data structure of choice for $k > 3$. Each split in a k -d tree uses only one dimension, which provides the flexibility to more efficiently partition the space for certain data distributions (e.g., points along the diagonal of the space). At each level of a k -d tree, only a single comparator is needed to determine which branch to take. In contrast, all comparators must be used in the k -dimensional extension of a quad tree. Also, deletion is much less complex in a k -d tree than in a quad tree.

Weaknesses: In general, if there are k dimensions, then we may think of each group of k consecutive levels as having a combined branching factor of 2^k , making comparisons along all k dimensions. In a k -dimensional extension of a quad tree, these k levels are compressed into one. However, to do this compression, a single k -dimensional point is used to partition the domain into 2^k subdomains. Thus the expected height of a k -dimensional extension of quad tree is k times less than the expected height of a k -d tree. While the expected number of comparisons is the same, the larger expected depth in a k -d tree slightly increases the search time. Furthermore, if there is support to make up to 3 comparisons in parallel, then a quad tree (for $k = 2$) or an oct tree (for $k = 3$) leads to significantly faster expected search times since all k comparisons can be made in parallel whereas for a k -d tree the comparisons must be made sequentially.

Critical Mutators: *remove*

Competing Data Structures: For two-dimensional data consider a quad tree (Chapter 48) and for three-dimensional data consider an oct tree, especially if there is support to evaluate the comparator for each dimension in parallel. For a one-dimensional spatial collection involving a range search, consider a skip list (Chapter 38). To perform a range search on a skip list, one would position a marker at the predecessor of the low end of the range and then advance through the elements (or tags) in the desired range.

If it is more natural for the application to associate with each element a tag that is a k -dimensional point (for $k \geq 2$), and it is uncommon for multiple elements to have the same tag, then a tagged k -d tree (Section 49.11.2) or a tagged bucket k -d tree should be considered.

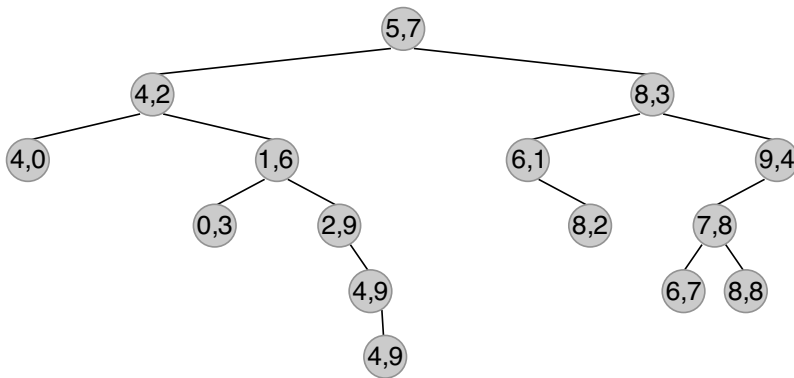


Figure 47.1

A populated example of a k -d tree, where each element is a two-dimensional point.

47.1 Internal Representation

The internal representation for a k -d tree uses `KDTreeImpl`, which is a special type of binary search tree data structure. In a standard binary search tree (Chapter 32), all elements are ordered by the comparator along a single dimension. Recall that the search in a binary search tree compares the target element to the element in the current node, with the search proceeding to the left child if the target is less, and proceeding to the right child if the target is greater. `KDTreeImpl` extends the concept of a binary search tree to support searching along multiple dimensions through the use of a special kind of comparator, the **alternating comparator** (Section 47.3), that can make comparisons of elements along any of a fixed number of dimensions.

As in a standard binary search tree, the `KDTreeImpl` searches down the tree by comparing the target element to the element in the current node. However, during the search, comparisons are made along different dimensions based on the current node in the path. More specifically, each node in the tree has a level number that corresponds to its distance from the root, and this level number determines which of the k comparators should be used to select whether to move to the left or right child. For example, suppose the comparator supports making comparisons along two different dimensions, x and y . Then the comparator would make comparisons along the x dimension at the even-numbered levels in the tree, and it would make comparisons along the y dimension at the odd-numbered levels. Figure 47.1 shows an example of such a tree, where each node contains a 2-dimensional point. Figure 47.2 shows the partition of the domain corresponding to this tree.

As elements are added to the tree, the alternating comparator cycles through the dimensions to make decisions at each level. Figure 47.3 shows the tree that results from inserting an element (1,6) into Figure 47.1. At the root (5,7), we use the x dimension and proceed left because $1 < 5$. At the next level, we use the y dimension for comparison with (4,2) and proceed right because $6 > 2$. This brings us to an existing element (1,6), and we proceed to the right because elements on the left must be strictly less. We reach node (2,9) at level 3 in the tree, where we use the y dimension and proceed left since $6 < 9$. At this point, a frontier node is reached and the duplicate element (1,6) is inserted.

In general, if there are k dimensions, then we may think of each group of k consecutive levels as having a combined branching factor of 2^k , making comparisons along all k dimensions. The standard binary search tree falls out as a special case of this, where $k = 1$ and the branching factor is 2. Because $k = 2$ is such a common case, the `QuadTree` (Chapter 48) provides special support for

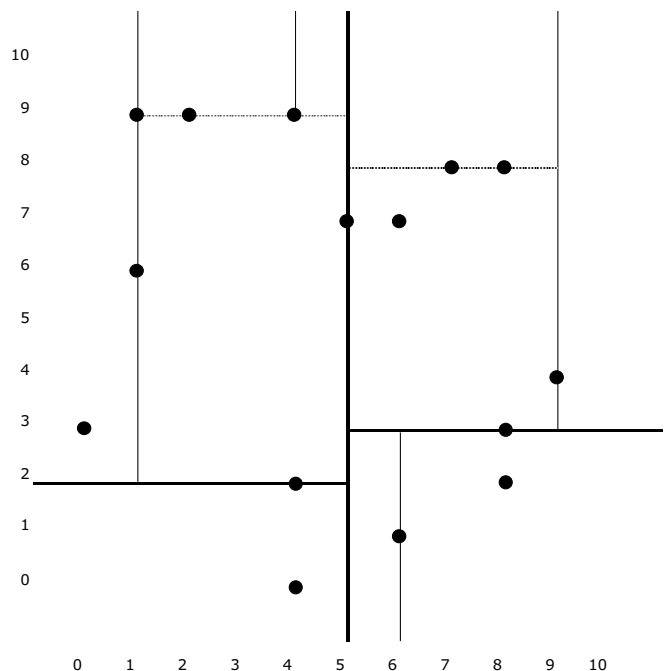


Figure 47.2
The partition of the domain corresponding to the k -d tree of Figure 47.1. The thickest line corresponds to the root, the next thickest line corresponds to the two nodes at level 1, and so on.

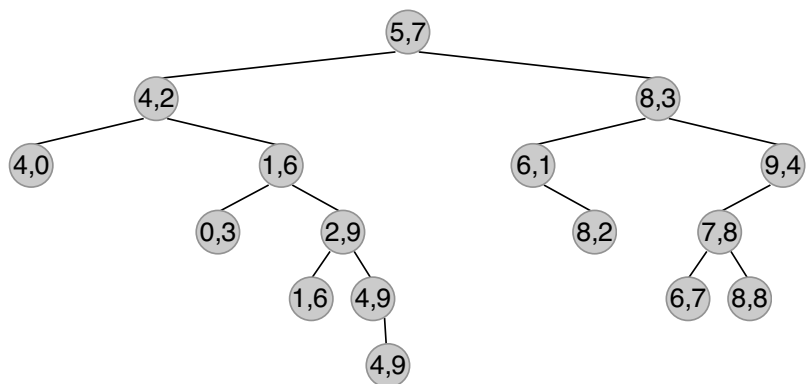


Figure 47.3
After insertion of a duplicate element (1,6) into the tree of Figure 47.1.

such spatial collections, with each node having a branching factor of 4. Having a level of the tree corresponding to each dimension provides flexibility to independently partition the two subdomains defined by each node.

Instance Variables and Constants: Each `KDTree` instance wraps a `KDTreeImpl` data structure (presented in Section 47.5). Nearly all methods delegate to the wrapped object.

`KDTreeImpl<E> tree;`

Populated Example: Figure 47.1 illustrates a populated example of a `KDTree`. Each node contains one element and the ordering is determined by an alternating comparator (Section 47.3) that cycles through the dimensions at each level in the tree.

Terminology:

- For a given tree, k denotes the number of different orderings, where each ordering corresponds to a different comparator. The constructor allows an arbitrary k comparators over the elements to be provided. We view each element as a point in a k -dimensional space.

- The **level** of a node x in a rooted tree is defined recursively as:

$$\text{level}(x) = \begin{cases} 0, & \text{if } x \text{ is the root} \\ \text{level}(x.\text{parent}) + 1 & \text{otherwise} \end{cases}$$

- The function $d(x) \in \{0, \dots, k-1\}$ denotes the dimension used to make comparisons with the element at node x during search. In particular, for node x , we define $d(x) \equiv \text{level}(x) \% k$.
- The term **discriminator** refers to the choice of which dimension is used to make comparisons at a given node. That is, the discriminator of x is dimension $d(x)$.
- For $i \in \{0, \dots, k-1\}$, the symbols $\leq_i$, $<_i$, $=_i$, $>_i$, and $\geq_i$ denote comparisons between two values with respect to dimension i of the comparator.
- For tree node x , we use $x.\text{data}$ to refer to the element held in x .
- For an element e , any subtree T , and a dimension $i \in \{0, \dots, k-1\}$, we say that $e \leq_i T$ if and only if for all $x \in T$, $e \leq_i x.\text{data}$. Likewise, we say that $e \geq_i T$ if and only if for all $x \in T$, $e \geq_i x.\text{data}$.

Abstraction Function: Let KDT be a `KDTree` instance. The abstraction function is that of the wrapped binary search tree $KDT.\text{tree}$. That is,

$$AF(KDT) = AF(KDT.\text{tree})$$

Design Notes: One may wonder why it is necessary to separate the `KDTreeImpl` class from the `KDTree` class itself, for it would seem natural for the `KDTree` class to directly implement the data structure. The dichotomy is necessary to resolve two competing design objectives. On one hand, the structure of a k -d tree is similar to a binary search tree, so it is desirable to extend the `BinarySearchTree` class. On the other hand, a k -d tree is technically not a binary search tree, and it would not make sense for a k -d tree to provide methods like *min*, *max*, *predecessor*, and *successor* because these are not uniquely defined unless the dimension of comparison is specified. By letting `KDTree` wrap a `KDTreeImpl` that extends `BinarySearchTree`, the desired structure is inherited without exposing the `BinarySearchTree` methods to the user of a k -d tree.

Optimizations: As search proceeds down the tree, the comparator cycles through the dimensions as it makes comparisons at each level. Consequently, it is not necessary to explicitly store the discriminator in each node. The *remove* method, however, does need to know the discriminator of the node being removed. The provided implementation computes this by determining the node's distance from the root. To save time in *remove*, one could allocate space for an extra variable in each node to store the discriminator explicitly. However, if finding the node to be removed already involves traversing a path through the tree from the root to that node, computing the distance to the root does not affect the asymptotic time complexity of the remove operation.

To avoid the (relatively insignificant) overhead and design complexity of delegating each public method call to the *KDTreeImpl*, one could let *KDTree* extend *BinarySearchTree* and throw an unsupported operation exception from methods such as the no argument *min*, *max*, *successor*, and *predecessor* that do not make sense for a multidimensional spatial collection. Alternatively, one could override some of these methods to use dimension zero as the default basis for comparison.

47.2 Representation Properties

We inherit all of the *BinarySearchTree* properties, but use a specialized definition of *INORDER* for a *k*-d tree to take into account the fact that comparisons are made along different dimensions at different levels of the tree.

INORDER: For all nodes x in the collection, $T(x.left) <_{d(x)} x.data \leq_{d(x)} T(x.right)$.

47.3 Alternating Comparator

package collection.spatial

AlternatingComparator<*E*> implements *Comparator*<*E*>

The alternating comparator manages the various comparators used by the *k*-d tree. To create an alternating comparator, two or more comparators are provided to the constructor, which stores them in an array whose indices correspond to each dimension's index. The alternating comparator cycles through the dimensions, advancing to the next dimension with each call to the *compare* method. The *reset* method positions the alternating comparator so that its next comparison is made along dimension 0. For convenience, the alternating comparator also provides a *compare* method that makes comparisons along a specified dimension.

```
protected Comparator<? super E>[] comparators;    //one comparator for each dimension
private int discriminator;                          //the index of the current dimension in the cycle
```

The constructor takes *comparators*, a variable number of comparators defining the dimensions of this comparator. The constructor requires that at least two comparators are provided.

```
public AlternatingComparator(Comparator<? super E> ... comparators) {
    if (comparators.length < 2)
```

```

        throw new IllegalArgumentException("at least two comparators are required");
        this.comparators = comparators;
        reset();
    }

```

The *reset* method returns the comparator to the beginning of the cycle of dimensions, so that the next dimension used for unqualified comparison will be dimension 0.

```

    public void reset() {
        discriminator = -1;
    }

```

There are two *compare* methods, **qualified** and **unqualified**, depending on whether or not they take a specific dimension for comparison as a parameter. The unqualified *compare* method takes as parameters *a*, the reference element, and *b*, the element to be compared against the reference. Calling the compare method has the side-effect of advancing this alternating comparator to the next dimension in the cycle, wrapping around to dimension 0 as needed. The comparator then determines the relative ordering of two elements on the basis of that dimension of comparison. The method returns a negative value if $a < b$ along the dimension of comparison, zero if *a* and *b* are equal along all dimensions, and a positive value if $a > b$ along the dimension of comparison. It is important to note that two values are deemed equal only if they match along *all* dimensions. If they match along the current dimension in the cycle but differ in another dimension, then a positive value is returned. This semantics is convenient for the KDTree implementation, in which ties along a node's dimension of comparison should result in traversal of the right subtree. It is assumed that the first parameter is always the reference point (e.g., the element in the tree) and the second parameter is the element under consideration (e.g., the one to be placed to its left or right).

It is important to note that this unqualified compare method violates the usual symmetric convention for comparators when a tie occurs along the dimension of comparison. If the elements passed to this compare method are equal with respect to the current dimension of comparison, but differ in some other dimension, then the method returns a positive value, regardless of the order in which the actual parameters are passed to the method.

```

    public int compare(E a, E b) {
        discriminator = nextDiscriminator(discriminator);           //advance in the cycle
        int result = compare(a, b, discriminator);
        if (result == 0) {
            for (int i = 0; i < comparators.length; i++)
                if (i != discriminator && compare(a, b, i) != 0)
                    return 1;
        }
        return result;
    }

```

The qualified *compare* method determines the relative ordering of two elements along a specified dimension. The method takes as parameters *a*, the reference element, *b*, the element to be compared against the reference, and *discriminator*, the index of the dimension along with the elements should be compared. The method returns a negative value if $a < b$, zero if $a = b$, and a positive value if $a > b$ with respect to the specified dimension. The other dimensions are ignored.

```

    public int compare(E a, E b, int discriminator) {
        return comparators[discriminator].compare(a, b);
    }

```

The *noGreaterThan* method takes as parameters *a*, the candidate element and *b*, the boundary element. It returns true if and only if the candidate is less than or equal to the boundary along all dimensions. Two calls to this method, using the maximum and minimum corners of a multidimensional bounding box, are sufficient to determine whether a given element lies within the box. (See the *withinBounds* method on [page 748](#).)

```
public boolean noGreaterThan(E a, E b) {
    for (int i = 0; i < comparators.length; i++) {
        if (compare(a, b, i) > 0)
            return false;
    }
    return true;
}
```

The *equalTo* method takes as parameters *a*, the candidate element and *b*, the target element. The method returns true if and only if the candidate is equal to the target along all dimensions.

```
public boolean equalTo(E a, E b) {
    for (int i = 0; i < comparators.length; i++) {
        if (compare(a, b, i) != 0)
            return false;
    }
    return true;
}
```

The method *getLastDiscriminatorUsed* returns the index of the most recently used dimension in this comparator's cycle, or -1 if the comparator has not made an unqualified comparison since it was last reset.

```
public int getLastDiscriminatorUsed() {
    return discriminator;
}
```

The method *nextDiscriminator* takes *discriminator*, the index of a dimension of this comparator and returns the index of the next dimension on the cycle.

```
public int nextDiscriminator(int discriminator) {
    return (discriminator+1) % comparators.length;
}
```

The method *getNumDimensions* returns the number of dimensions used by this comparator, which is equal to the number of comparators provided to the constructor.

```
public int getNumDimensions() {
    return comparators.length;
}
```


47.4 KNode Inner Class

BSTNode

↑ **KNode**

The KNode class is an inner class of the KDTreeImpl class (Section 47.5). The KNode class extends the BSTNode inner class of the binary search tree to take into account multiple dimensions along which elements may be compared, which affects methods such as *min* and *max*, as well as the algorithm for *remove*. The constructor takes *data*, the element to hold in the node.

```
KNode(E data) {
    super(data);
}
```

The *getData* method returns the element held in the node.

```
E getData() {
    return data;
}
```

The *getDiscriminator* method takes no parameters and returns the dimension along which search comparisons should be made with the element in this node.

```
protected int getDiscriminator() {
    int depth = 0;
    KNode x = this;
    while (x.parent != null) {
        x = (KNode) x.parent;
        depth++;
    }
    int k = ((AlternatingComparator) KDTreeImpl.this.comp).getNumDimensions();
    return depth % k;
}
```

Correctness Highlights: The algorithm traverses the path to the root, measuring the depth of the node in the tree. Since there are k dimensions, $depth \% k$ is the correct discriminator. ■

The *min* method takes as parameters two KNodes *a* and *b*, and *dimension*, the dimension along which *a* and *b* should be compared. It returns the node *a* or *b* whose element is less according to the given discriminator. When they are equal with respect to the given dimension, node *b* is returned arbitrarily. If node *b* is a frontier node, node *a* is returned.

```
protected final KNode min(KNode a, KNode b, int dimension) {
    if (b.isFrontier() ||
        ((AlternatingComparator<E>) comp).compare(a.data, b.data, dimension) < 0)
        return a;
    else
        return b;
}
```

The *minimumInSubtree* method takes as parameters a *discriminator*, the dimension to be used for making comparisons, and the *nodeDiscriminator*, the discriminator used during search at the level of this node. The method returns a node containing a minimum element in the subtree rooted at this node, where comparisons are made with respect to the given discriminator. The method requires that the parameter value of *nodeDiscriminator* is the correct discriminator for this node's level in the tree. That is, if this method is called on node x , *nodeDiscriminator* must have the value $d(x)$.

```
protected KNode minimumInSubtree(int discriminator, int nodeDiscriminator) {
    boolean goLeft = (discriminator == nodeDiscriminator);
    if (isFrontier() || ( goLeft && ((KNode) left).isFrontier()))
        return this;
    int nextLevelDiscriminator =
        ((AlternatingComparator<E>) comp).nextDiscriminator(nodeDiscriminator);
    KNode leftMin =
        ((KNode) left).minimumInSubtree(discriminator, nextLevelDiscriminator);
    if (goLeft)
        return leftMin;
    KNode rightMin =
        ((KNode) right).minimumInSubtree(discriminator, nextLevelDiscriminator);
    return min( min(this, leftMin, discriminator), rightMin, discriminator);
}
```

Correctness Highlights: At node x , if $d(x)$ is the dimension along which the minimum is to be found, then by INORDER we know the entire right subtree can be ignored, so search proceeds to the left, taking this node as the minimum if it has no left child. However, if x has a different discriminator, then the entire subtree must be searched (recursively), and the minimum among this node's value and the minimum values of its two subtrees is returned. Note that the next level discriminator is correctly advanced with each recursive call down the tree. ■

The *minimumInSubtree* method takes no parameters and returns a node containing a minimum element in the subtree rooted at this node, where comparisons are made with respect to the discriminator of the node. This method requires that the tree's comparator was last used to compare a node just above this level, as would be the case if this node's parent had just been found for removal.

```
protected KNode minimumInSubtree() {
    AlternatingComparator<E> comparator = (AlternatingComparator<E>) comp;
    int discriminatorAtRoot = comparator.getLastDiscriminatorUsed();
    return minimumInSubtree(discriminatorAtRoot,
        comparator.nextDiscriminator(discriminatorAtRoot));
}
```

Correctness Highlights: Follows from the correctness of the previous *minimumInSubtree* method. That method takes as parameters this node's discriminator and the discriminator at the next level. ■

The *max* method takes as parameters two KNodes a and b , and *dimension*, the dimension along which a and b should be compared. It returns the node a or b whose element is greater according to the given dimension. When node b is a frontier node or the nodes are equal with respect to the given dimension, node a is returned.

```

protected final KNode max(KNode a, KNode b, int dimension) {
    if (b.isFrontier() ||
        ((AlternatingComparator<E>) comp).compare(a.data, b.data, dimension) ≥ 0)
        return a;
    else
        return b;
}

```

The *maximumInSubtree* method takes as parameters a *discriminator*, the dimension to be used for making comparisons and the *nodeDiscriminator*, the discriminator used during search at the level of the current node. The method returns a node containing a maximum element in the subtree rooted at this node, where comparisons are made with respect to the given discriminator. The method requires that the parameter value of *nodeDiscriminator* is the correct discriminator for this node's level in the tree.

```

protected KNode maximumInSubtree(int discriminator, int nodeDiscriminator) {
    boolean goRight = (discriminator == nodeDiscriminator);
    if (isFrontier() || (goRight && ((KNode) right).isFrontier()))
        return this;
    int nextLevelDiscriminator =
        ((AlternatingComparator<E>) comp).nextDiscriminator(nodeDiscriminator);
    KNode rightMax =
        ((KNode) right).maximumInSubtree(discriminator, nextLevelDiscriminator);
    if (goRight)
        return rightMax;
    KNode leftMax =
        ((KNode) left).maximumInSubtree(discriminator, nextLevelDiscriminator);
    return max( max(this, leftMax, discriminator), rightMax, discriminator);
}

```

| *Correctness Highlights:* Analogous to the similar *minimumInSubtree* method. ■

The *maximumInSubtree* method takes no parameters and returns a node containing a maximum element in the subtree rooted at this node, where comparisons are made with respect to the discriminator of the node.

```

protected KNode maximumInSubtree() {
    AlternatingComparator<E> comparator = (AlternatingComparator<E>) comp;
    int discriminatorAtRoot = comparator.getLastDiscriminatorUsed();
    return maximumInSubtree(discriminatorAtRoot,
        comparator.nextDiscriminator(discriminatorAtRoot));
}

```

The *remove* method takes as its parameter *disc*, the discriminator used during search at the level of the current node. The method removes this node from the tree, restructuring the tree as needed to preserve the INORDER property. The method requires the parameter to be the correct discriminator for this node's level in the tree.

```

protected void remove(int disc) {
    KNode right = (KNode) this.right;
    KNode left = (KNode) this.left;

    if (right.isFrontier() && !left.isFrontier()) {           //special case: Only the right is empty
        this.right = right = left;                             //move the only subtree to the right side
        this.left = FRONTIER.L;
    }
    if (!right.isFrontierNode()) {                             //Case 1: There is a right subtree
        KNode replacement = right.minimumInSubtree(disc,
            ((AlternatingComparator) comp).nextDiscriminator(disc));
        KDTreeImpl.this.remove(replacement);                 //remove min from right subtree
        substituteNode(replacement);                           //replace this node by the minimum
    } else {                                                  //Case 2: This is a leaf node
        substituteNode(FRONTIER.L);
    }
}

```

Correctness Highlights: Let x be this node, and let e be its contained element. Recall that $d(x)$ is discriminator for this node's level, $level(x)$, in the tree. If x is a leaf, it is simply replaced by a frontier node and INORDER is preserved. Otherwise, before node x is removed, we know by INORDER that all elements in the left subtree of x (if any) are strictly less than its element, with respect to $d(x)$. Similarly, we know e is no greater, with respect to $d(x)$, than the elements in the right subtree of x . More formally, we know that $T(this.left) <_{d(x)} this.data$ and $this.data \leq_{d(x)} T(this.right)$. If x is not a leaf, x is replaced by the minimum node in its right subtree, with respect to $d(x)$. Because the replacement element is the minimum of the (old) right subtree, all nodes in the (new) right subtree are no smaller than the replacement node. By transitivity in the above inequalities, we know that since the replacement node came from the right subtree, it must be larger (with respect to $d(x)$) than all elements in the left subtree of x . Therefore, INORDER is preserved.

A special case arises when x has only a left subtree. In that case, the left subtree is moved to the right of x and remove proceeds normally. Moving this node's left subtree to its right side temporarily violates the INORDER property, but once x is replaced by the minimum in that subtree with respect to $d(x)$, the property is restored. ■

Figure 47.4 illustrates the results of the remove algorithm. First, element (9,4) is removed from the tree shown at the top of the figure. Note that since (9,4) has no right subtree, the special case in the remove algorithm applies, so its left subtree is moved to its right. The next step in the remove algorithm is to find a replacement node for (9,4), the minimum element in its right subtree. In finding the minimum element, the x dimension is used as the discriminator because (9,4) is at an even-numbered level in the tree. Therefore, the element (6,7) is chosen as the replacement and is recursively removed. Because (6,7) has no children, it is simply replaced by the node FRONTIER.L. After the recursive call, (6,7) is substituted for (9,4) in the tree, resulting in the second tree shown in the figure.

From the second tree, the root element (5,7) is removed. Again, the x dimension is used as the discriminator to find a minimum element in the right subtree. The element (6,7) is selected. It is recursively removed and then replaces the root, resulting in the bottom tree shown in the figure.

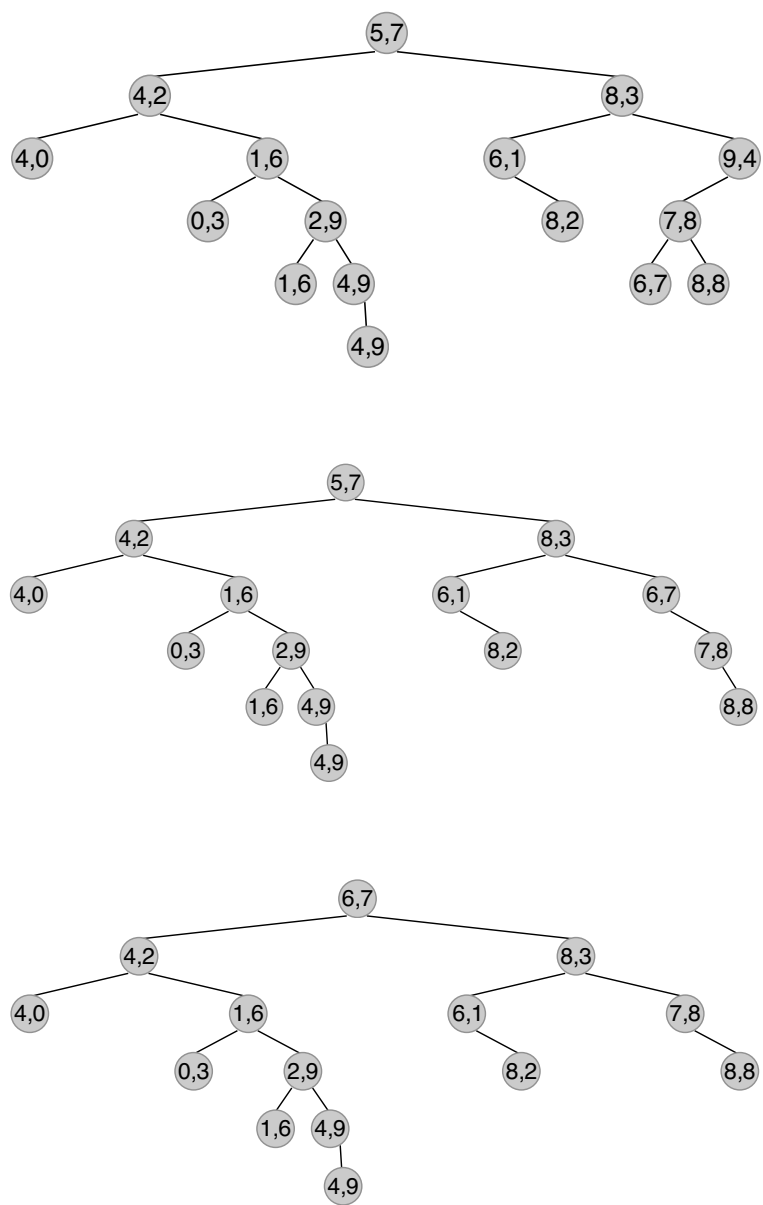


Figure 47.4
Removing elements (9,4) and subsequently (5,7) from a *k*-d tree.

47.5 KDTreeImpl Class

package collection.spatial

AbstractCollection<E> implements *Collection*<E>

└ BinarySearchTree<E> implements *OrderedCollection*<E>

└ **KDTreeImpl**<E> implements *SpatialCollection*<E>

In this section we describe the KDTreeImpl class. The only change in the internal representation for the KDTreeImpl class from the BinarySearchTree class is that a k -d node replaces the binary search tree node.

The constructor for a KDTreeImpl takes as its parameter a *comparator*, the alternating comparator for making comparisons. Recall that an alternating comparator cycles through each of a fixed number of dimensions, moving to the next dimension each time it is asked to make a comparison. In this way, the alternating comparator can be used as the basis for searching within a k -d tree using the same search algorithm as for a binary search tree. That is, since the comparator takes care of cycling through the various dimensions as the search proceeds down the tree, it is not necessary to replace the logic of the *find* method inherited from BinarySearchTree.

```
public KDTreeImpl(AlternatingComparator<E> comparator) {
    super(comparator);
}
```

The factory method *createTreeNode* takes as its parameter *data*, the element to be stored in the node and returns a newly created KNode containing the data.

```
protected KNode createTreeNode(E data) {
    return new KNode(data);
}
```

The method *resetComparator* is a convenience method to restart the comparator at dimension zero, in preparation for a new search down the tree.

```
protected void resetComparator() {
    ((AlternatingComparator<E>) comp).reset();
}
```

The method *minimum* takes as its parameter a *dimension*, the desired dimension along which the minimum should be found. The method returns a least element in the tree along the given dimension. It throws a *NoSuchElementException* when the tree is empty and throws an *IllegalArgumentException* when the given dimension index is not supported by the tree's comparator.

```
public E minimum(int dimension) {
    if (getSize() == 0)
        throw new NoSuchElementException();
    if (dimension < 0 ||
        dimension ≥ ((AlternatingComparator<E>) comp).getNumDimensions())
        throw new IllegalArgumentException("Illegal dimension " + dimension);
    return ((KNode) root).minimumInSubtree(dimension, 0).getData();
}
```

The method *maximum* takes as its parameter a *dimension*, the desired dimension along which the maximum should be found. The method returns a greatest element in the tree along the given dimension. It throws a *NoSuchElementException* when the tree is empty and throws a *IllegalArgumentException* when the given dimension index is not supported by the tree's comparator.

```
public E maximum(int dimension) {
    if (getSize() == 0)
        throw new NoSuchElementException();
    if (dimension < 0 ||
        dimension ≥ ((AlternatingComparator<E>) comp).getNumDimensions())
        throw new IllegalArgumentException("Illegal dimension " + dimension);
    return ((KDNode) root).maximumInSubtree(dimension, 0).getData();
}
```

The purpose of the method *withinBounds* is to perform an **orthogonal range search**, finding all elements in the structure whose values are within a certain bound. In this case, the range is defined by two opposite "corners." The method takes as its parameters a *minCorner*, the corner of the range having the values in all dimensions at the low end of the range and *maxCorner*, the corner of the range having the values in all dimensions at the high end of the range. The method returns a collection of all the elements in the tree that fall within the given range, inclusive. The method requires *minCorner* to be less than or equal to *maxCorner* along all possible dimensions of the comparator.

```
public Collection<E> withinBounds(E minCorner, E maxCorner) {
    Collection<E> elements = new DoublyLinkedList<E>();
    withinBounds((KDNode) root, minCorner, maxCorner, elements, 0);
    return elements;
}
```

The auxiliary method *withinBounds* recursively performs a range search on behalf of the public method. The method takes as its parameters a *node*, the root of a subtree in which to search, *minCorner*, the corner of the range having the values in all dimensions at the low end of the range, *maxCorner*, the corner of the range having the values in all dimensions at the high end of the range, *elements*, a collection to which the elements within the range should be added, and *dimension*, the discriminator of the given node. The method adds to the given collection all elements in the subtree that fall within the given range, inclusive. The method requires *minCorner* to be less than or equal to *maxCorner* along all possible dimensions of the comparator, and that the given dimension be the correct discriminator for the node according to its level in the tree.

```
private void withinBounds(KDNode node, E minCorner, E maxCorner,
    Collection<? super E> elements, int dimension) {
    if (node.isFrontierNode())
        return;
    E element = node.getData();
    AlternatingComparator<E> comp = (AlternatingComparator<E>) this.comp;
    int minCompare = comp.compare(element, minCorner, dimension);
    int maxCompare = comp.compare(element, maxCorner, dimension);

    if (minCompare ≥ 0 && maxCompare ≤ 0) {
        if (comp.noGreaterThan(minCorner, element) &&
            comp.noGreaterThan(element, maxCorner))
            //candidate
```

```

        elements.add(element);
    }
    int nextDim = comp.nextDiscriminator(dimension);
    if (minCompare ≥ 0)
        withinBounds(node.getChild(0), minCorner, maxCorner, elements, nextDim);
    if (maxCompare ≤ 0)
        withinBounds(node.getChild(1), minCorner, maxCorner, elements, nextDim);
}

```

Correctness Highlights: At each node x , the algorithm must decide if the element e in x is within the bounding box, and then must decide which portion(s) of the subtree may contain elements within the bounding box. To support both decisions, e is compared to the corners of the bounding box along the dimension $d(x)$. If e is found to fall between the corners, then e is a candidate and is compared to the bounding box corners along all dimensions, and added to the collection of elements if it is found to be within the bounding box. The comparisons of e with the minCorner and maxCorner along dimension $d(x)$ are also used to determine which portions of the subtree rooted at this node must be searched. By INORDER, we know that it is necessary to consider elements in the left subtree only if $e \geq_{d(x)} \text{minCorner}$. Similarly, we know it is necessary to consider elements in the right subtree only if $e \leq_{d(x)} \text{maxCorner}$. ■

The *find* and *findLastInsertPosition* methods use the same algorithm as binary search tree, and are overridden here only to reset the alternating comparator so each search begins from the root using dimension zero.

We override the *find* method to reset the comparator before use by other inherited methods, such as *contains* and *remove*. The method takes as its parameter *element*, the element to be found. If the tree contains a given element, the method returns a reference to a tree node that contains an occurrence of *element*. If *element* is not in the collection, then an insert position is returned, with the *parent* field of the frontier node set to the node that preceded it on the search path.

```

protected KNode find(E element) {
    resetComparator();
    return (KNode) super.find(element);
}

```

We override the *findLastInsertPosition* method to reset the comparator before use by other inherited methods, such as *add* and *addTracked*. It takes as its parameter *element*, the element to be found. If the tree contains a given element, the method returns a reference to a tree node that contains an occurrence of *element*. If *element* is not in the collection, then an insert position is returned, with the *parent* field of the frontier node set to the node that preceded it on the search path.

```

protected BSTNode findLastInsertPosition(E element) {
    resetComparator();
    return super.findLastInsertPosition(element);
}

```

The method *remove* method takes *element*, the element to remove. It returns *true* if the element is in the tree. Otherwise, *false* is returned.

```

public boolean remove(E element) {
    if (super.remove(element)) {

```



```

        version.increment();
        return true;
    }
    return false;
}

```

The method *remove* removes a given *KDNode* from the tree. Its only parameter is *x*, the node to be removed. All work is delegated to the tree node itself.

```

protected void remove(TreeNode x) {
    KDNode toDelete = (KDNode) x;
    toDelete.remove(toDelete.getDiscriminator());
}

```

47.6 KD-Tree Methods

In this section we describe the methods of the *KDTree* class. Most of these methods delegate to the corresponding *KDTreeImpl* method.

Constructors

The constructor accepts *comparators*, a variable number of comparators defining the dimensions of this spatial collection. The constructor requires that at least two comparators are provided.

```

public KDTree(Comparator<? super E> ... comparators) {
    super(new AlternatingComparator<E>(comparators));
    tree = new KDTreeImpl<E>((AlternatingComparator<E>) comp);
}

```

TrivialAccessors

The *getSize* method returns the size of the collection, which is simply the number of elements in the wrapped tree.

```

public int getSize() {
    return tree.getSize();
}

```

Algorithmic Accessors

The *equivalent* method takes *a*, the first element to compare, and *b*, the second element to compare. It returns *true* if and only if *a* and *b* are equivalent. Note that, in the case of a multidimensional data set, equivalence is defined as the elements having equal value along all dimensions.

```

protected boolean equivalent(E a, E b) {
    return ((AlternatingComparator<E>) comp).equalTo(a, b);
}

```

Because the data set is multidimensional, the inherited *compare* method does not make sense. Comparison cannot occur without specifying a particular dimension. Therefore, it throws an *UnsupportedOperationException*.

```

protected int compare(E o1, E o2) {
    throw new UnsupportedOperationException();
}

```

The *min* method takes *dimension*, the dimension along which elements should be compared, and returns a least element in the collection along the given dimension. The work is delegated to the wrapped tree.

```

public E min(int dimension) throws NoSuchElementException {
    return tree.minimum(dimension);
}

```

The *max* method takes *dimension*, the dimension along which elements should be compared, and returns a greatest element in the collection along the given dimension. Again, the work is delegated to the wrapped tree.

```

public E max(int dimension) {
    return tree.maximum(dimension);
}

```

The *contains* method takes *element*, an element that may or may not be in the collection, returns *true* if and only if an equivalent element exists in the collection. The work is delegated to the wrapped tree.

```

public boolean contains(E element) {
    return tree.contains(element);
}

```

The *withinBounds* method takes as its parameters a *minCorner*, the corner of the range having the values in all dimensions at the low end of the range and *maxCorner*, the corner of the range having the values in all dimensions at the high end of the range. The method returns a collection of all the elements in the tree that fall within the given range. The method requires *minCorner* to be less than or equal to *maxCorner* along all possible dimensions of the comparator.

```

public Collection<E> withinBounds(E minCorner, E maxCorner) {
    return tree.withinBounds(minCorner, maxCorner);
}

```

The method *getLocator* takes *x*, the element to track. It returns a tracker that has been initialized at the given element. It throws a *NoSuchElementException* when the given element is not in the collection. The work is delegated to the wrapped tree, which in turn inherits the implementation from *BinarySearchTree*.

```

public Locator<E> getLocator(E x) {
    return tree.getLocator(x);
}

```

Content Mutators

The *add* method takes *element*, an item to be added to the collection. The work is delegated to the wrapped tree.

```
public void add(E element) {
    tree.add(element);
}
```

The *addTracked* method takes *element*, an item to be added to the collection and returns a tracker positioned at that element. The work is delegated to the wrapped tree.

```
public Locator<E> addTracked(E element) {
    return tree.addTracked(element);
}
```

The public *remove* method takes *element*, the element to remove. It removes from the collection an arbitrary element (if any) equivalent to *element*. Note that equivalence is defined as being of equal value along all dimensions of the comparator. It returns *true* if an element was removed, and *false* otherwise.

```
public boolean remove(E element) {
    return tree.remove(element);
}
```

Locator Initializers

The *iterator* method creates a new tracker at FORE. Note that because the data set is multidimensional, there is not a well-defined ordering of the elements. Therefore, one should not rely upon any particular ordering of the data during iteration. The work is delegated to the tree, which inherits it from the binary search tree locator.

```
public Locator<E> iterator() {
    return tree.iterator();
}
```

The *iteratorAtEnd* method creates a new tracker that is at AFT. Again, because the data set is multidimensional, there is not a well-defined ordering of the elements. Therefore, one should not rely upon any particular ordering during iteration. The work is delegated to the tree, which inherits it from *BinarySearchTree*.

```
public Locator<E> iteratorAtEnd() {
    return tree.iteratorAtEnd();
}
```

47.7 Performance Analysis

The asymptotic time complexities of all public methods for the *KDTree* class are shown in [Table 47.5](#). The asymptotic time complexities for all of the public methods of the *BinarySearchTree* Tracker class (which is used here) are repeated for convenience in [Table 47.6](#). If the elements inserted are randomly selected, then the expected value of h_x for a randomly selected x is about $2 \ln n \approx 1.382 \log_2 n$. However, in the worst case $h_x = n$.

method	time complexity
<i>ensureCapacity(x)</i>	$O(1)$
<i>iterator()</i>	$O(1)$
<i>trimToSize()</i>	$O(1)$
<i>add(o),addTracked(o)</i>	$O(h)$
<i>contains(o)</i>	$O(h)$
<i>getEquivalentElement(o)</i>	$O(h)$
<i>getLocator(o)</i>	$O(h)$
<i>min(int d)</i>	$O(h)$
<i>max(int d)</i>	$O(h)$
<i>remove(o)</i>	$O(h)$
<i>withinBounds(minCorner,maxCorner)</i>	$O(n^{(k-1)/k} + h + r)$
<i>accept(v)</i>	$O(n)$
<i>clear()</i>	$O(n)$
<i>toArray()</i>	$O(n)$
<i>toString()</i>	$O(n)$
<i>retainAll(c)</i>	$O(n(c + h))$
<i>addAll(c)</i>	$O(c \log(n + c))$

Table 47.5 Summary of the asymptotic time complexities for the k -d tree where k is the number of dimensions, and h is the height of the tree. In the worst case $h = n$. If the elements are randomly generated, the expected height is approximately $1.386 \log_2 n$. For the orthogonal range search (*withinBounds*) r is the number of elements returned.

locator method	time complexity
constructor	$O(1)$
<i>get()</i>	$O(1)$
<i>advance()</i>	$O(h)$
<i>hasNext()</i>	$O(h)$
<i>next()</i>	$O(h)$
<i>remove()</i>	$O(h)$
<i>retreat()</i>	$O(h)$

Table 47.6 Summary of the amortized time complexities for the public locator methods of the binary search tree tracker class used by the k -d tree.

The expected search time for a k -d tree constructed from inserting n elements in a random order is $O(\log n)$. In the worst-case, the tree has linear height. The rest of the analysis follows from that of the wrapped binary search tree method. More specifically, Bentley [21] proved that the probability of constructing a particular tree structure by inserting n random elements into an empty k -d tree is the same as the probability of obtaining that tree by random insertions into a standard binary search tree. Thus all results known about the structure of a standard binary search tree apply to k -d trees. In particular, it follows from the work of Knuth [97] that for C_n , the number of nodes visited during a search in a k -d tree, $E[C_n] = 2(1 + 1/n)H_n - 3$, where $H_n = 1 + 1/2 + \dots + 1/n$. So the expected cost to insert an element in a k -d tree, and to locate a desired element, is approximately $1.386 \log_2 n$.

Bentley also showed that deletion of the root has $O(n^{(k-1)/k})$ time complexity, and deleting a randomly selected node has expected logarithmic cost (independent of k). Each orthogonal range query examines at most $O(n^{(k-1)/k} + r)$ leaves, where r is the number of elements returned. Observe that each node in the k -d tree corresponds to a (possibly open) k -dimensional axis-aligned box that contains all the points in the subtree rooted at x . The root corresponds to the entire domain. For node x with non-null parent p , the region corresponding to x is the portion of the region corresponding to p on the side associated with x of the halfspace of the discriminator defined at p . In the remainder of this paragraph, we treat a node and its corresponding region interchangeably. The cost of an orthogonal range search for k -dimensional axis-aligned box b is dominated by the number of nodes that are examined in order to find the nodes that define the roots of the subtrees that contained entirely within b . Observe that the nodes that must be examined are those that are intersected by one of the $2k$ faces of b . We can upperbound the number of nodes that intersect a face f of b by the number of nodes that intersect the plane defined by this face. Consider node x and the 2^k regions defined by the descendants $t_1, \dots, t_{2^k}$ that occur after k levels of splitting. Observe that the discriminator at one of these k levels is parallel to face f , so at most 2^{k-1} of the regions $t_1, \dots, t_{2^k}$ can intersect f . Let $Q(n)$ be the number of regions examined by an orthogonal range search for a region holding n elements. It can be shown that the expected number of elements in t_i (for $i = 1, \dots, 2^k$) is $n/2^k$. In addition, k nodes are examined in the k levels examined. Thus, we get the recurrence that $Q(1) = \Theta(1)$ and $Q(n) = 2^{k-1}Q(n/2^k) + k$. For k a constant, using the master method (Section B.6), we obtain that $Q(n) = \Theta(n^{(k-1)/k})$. Since b has $2k$ faces, we obtain an upperbound of $2k \cdot O(n^{(k-1)/k})$. Finally, it takes $O(h+r)$ time to traverse the subtrees contained within b where h is the expected height of the tree, which is approximately $1.386 \log_2 n$. Thus, for k constant, the expected cost for an orthogonal range query is $O(n^{(k-1)/k} + h + r) = O(n^{1-\frac{1}{k}} + r)$ for $k \geq 2$. So for $k = 2$, the expected cost of *withinBounds* is $O(\sqrt{n} + r)$, and for $k = 3$, the expected cost of *withinBounds* is $O(n^{2/3} + r)$.

There are other queries that can be efficiently performed with a k -d tree. A **partial match query** is a query in which desired values for t of the k dimensions are specified, and the other dimensions can have any value. The number of elements that must be examined (for any k -d tree) for a partial match query with t specified values is $O(n^{(k-t)/k} + r)$ where r is the number of elements returned. Thus, the expected cost is $O(n^{(k-t)/k} + \log n + r)$. So for $k = 2$, a partial match query has expected time complexity $O(r + \sqrt{n})$. Freidman, Bentley and Finkel [67] showed that a k -d tree can be used to find the m nearest neighbors of a specified point by visiting $O(\log n)$ nodes and performing approximately $m2^k$ distance calculations.

47.8 Quick Method Reference

KDTree Public Methods

- p. 750 `KDTree(Comparator<? super E> ... comparators)`
 p. 98 `void accept(Visitor<? super E> v)`

- p. 751** void **add**(E element)
- p. 100 void **addAll**(Collection<? extends E> c)
- p. 752** Locator<E> **addTracked**(E element)
- p. 100 void **clear**()
- p. 97 boolean **contains**(E value)
- p. 99 void **ensureCapacity**(int capacity)
- p. 96 int **getCapacity**()
- p. 96 Comparator<? super E> **getComparator**()
- p. 98 E **getEquivalentElement** (E target)
- p. 751** Locator<E> **getLocator**(E x)
- p. 96 int **getSize**()
- p. 96 boolean **isEmpty**()
- p. 752** Locator<E> **iterator**()
- p. 752** Locator<E> **iteratorAtEnd**()
- p. 751** E **max**(int dimension)
- p. 751** E **min**(int dimension)
- p. 752** boolean **remove**(E element)
- p. 100 void **retainAll**(Collection<E> c)
- p. 97 Object[] **toArray**()
- p. 97 E[] **toArray**(E[] array)
- p. 98 String **toString**()
- p. 99 void **trimToSize**()
- p. 751** Collection<E> **withinBounds**(E minCorner, E maxCorner)

KDTree Internal Methods

- p. 97 int **compare**(E e1, E e2)
- p. 750** int **compare**(E o1, E o2)
- p. 750** boolean **equivalent**(E a, E b)
- p. 97 boolean **equivalent**(E e1, E e2)
- p. 99 void **traverseForVisitor**(Visitor<? super E> v)
- p. 98 void **writeElements**(StringBuilder s)

Chapter 48

Quad Tree Data Structure

package `collection.spatial`

`AbstractCollection<E>` implements *Collection<E>*

└─ `AbstractSearchTree<E>`

└─ `QuadTree<E>` implements *SpatialCollection<E>*

Uses: Java array and references, `DoublyLinkedList` (Chapter 16)

Used By: `TaggedQuadTree` (Section 49.11.3)

Strengths: The expected depth of a k -dimensional extension of a quad tree is a multiplicative factor of k less than a k -d tree (Chapter 47), since the branching factor is 2^k at each level of the tree. So for uniformly distributed points, a quad tree has 3 times fewer nodes than a 2d-tree, and 9 times fewer nodes than a 3d-tree. A quad tree (for $k = 2$) or an oct tree (for $k = 3$) leads to faster search times, especially if up to 3 comparisons can be made in parallel. Even when the comparisons are made sequentially, the number of references to follow is reduced by a factor of k when using a k -dimensional extension of a quad tree.

Weaknesses: Since each node in a k -dimensional extension of a quad tree has 2^d children, thus the cost to search and modify the tree grows exponentially in the number of dimensions. Thus, it is rare to consider anything except a quad tree ($k = 2$) and an oct tree ($k = 3$). For data that is clustered in a small area, a quad tree will have many empty regions, yielding poor space usage. For higher dimensions there tend to be many empty regions, making the space usage prohibitive. Also, all k comparators must be used to determine which branch to consider next, and deletion is quite complex for a quad tree.

Critical Mutators: *remove*

Competing Data Structures: Use a k -d tree (Chapter 48) if range searching in three or more dimensions is required. For one-dimensional spatial collections involving range search, consider a skip list (Chapter 38). To perform a range search on a skip list, one would position a marker at the predecessor of the low end of the range and then advance through the elements (or tags) in the desired range.

48.1 Internal Representation

Each node in a quad tree contains an element representing an (x, y) point. Each point divides the plane into four quadrants: lower left, lower right, upper right, and upper left. Each node has

four subtrees containing elements for points in these respective quadrants. For example, if a node contains an element for point (35, 50), then all elements in the upper right subtree of that node correspond to points whose x values are greater than 35 and whose y values are greater than 50.

Instance Variables and Constants: We define a special tree node class, called `QTreeNode`, to hold the elements of the quad tree. Each quad tree has a root node, which is initially the frontier tree node. Each quad tree has an `XYComparator` with a special method that determines in which quadrant a given point belongs, in relation to another given point.

```
final QTreeNode FRONTIER = createTreeNode(null);
QTreeNode root = FRONTIER;
XYComparator<? super E> comp;
```

The remaining instance variables are used by the (rather complicated) `remove` method. When an element is removed, a replacement point must be found to take its place in the tree, and then portions of the tree must be adjusted to maintain the required spatial relationships. Most of the following variables are used to find which replacement element is likely to result in the fewest necessary adjustments. After the replacement element has been selected, *dangerZone* is used to determine which other elements need to be moved to restore the required spatial relationships. The arrays and the box are allocated here to avoid creating them each time *remove* is called.

```
double[] xDistance = new double[4];
double[] yDistance = new double[4];
Object[] candidates = new Object[4];
Box dangerZone = new Box();
```

Populated Example: Figure 48.1 shows a populated example of a quad tree. The top representation shows the tree structure, with each node having up to four children. The bottom representation shows each node as a labeled point that divides its region into four quadrants. For example, notice that the root (100,150) divides the universe into four quadrants. Its leftmost child (50,60) further subdivides the root's lower left quadrant into four regions. Ties in x and y , respectively, go to the left and lower quadrants. For example, when (30,150) is inserted, it is placed in a left quadrant of the root because $30 < 100$ and is placed in a lower quadrant because its y value of 150 is equal to that of the root.

Terminology:

- Each element in a quad tree (implicitly) defines a 2-dimensional point. For an element a , we let $a.x$ refer to the x -coordinate of a , and let $a.y$ refer to the y -coordinate of a .
- For two elements a and b , we say that a is **left of** b iff $a.x \leq b.x$. Otherwise, we say that a is **right of** b .
- For two elements a and b , we say that a is **below** b iff $a.y \leq b.y$. Otherwise, we say that a is **above** b .
- With respect to each node in a quad tree, we number each of the four quadrants relative to its contained point:
 - lower left (0)
 - lower right (1)
 - upper right (2)
 - upper left (3)

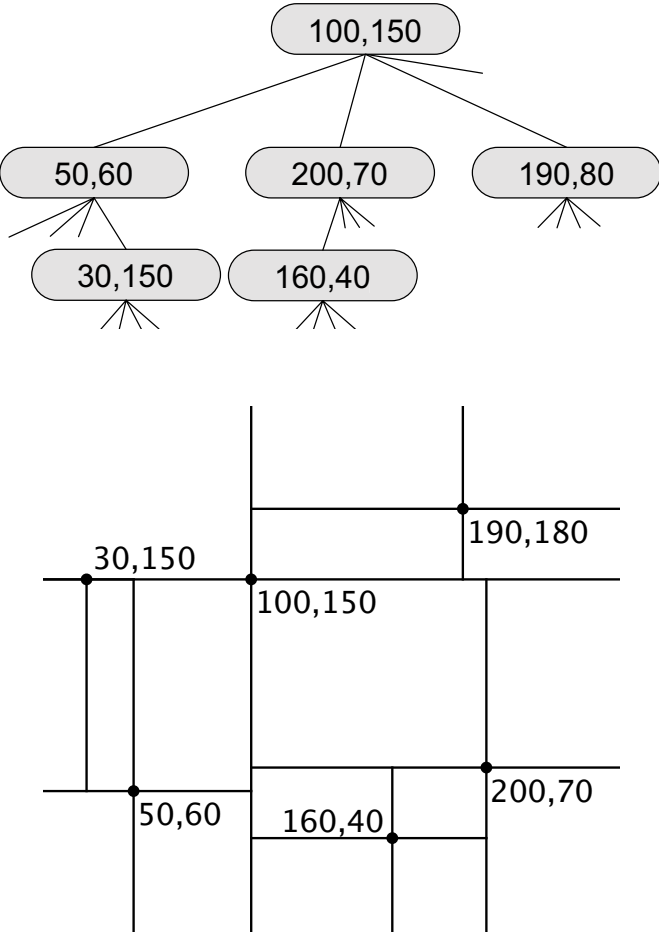


Figure 48.1
A populated example of a quad tree (above), with corresponding labeled points and spatial quadrants (below).

In other words, quadrants 0 and 1 contain points whose y coordinates are less than or equal to that of the origin, with other points going to quadrants 2 and 3. Similarly, quadrants 0 and 3 contain points whose x coordinates are less than or equal to that of the origin, with other points going to quadrants 1 and 2. For example, in [Figure 48.1](#), (160,40) is in quadrant 1 (the lower right quadrant) of the root node (100,150), but it is child 0 (in the lower left quadrant) of its immediate parent (200,70). The immediate children of each node are similarly numbered (*child(0)*, *child(1)*, etc.) to correspond with the numbered quadrants.

- The **quadrant** of a node is defined relative to its parent. That is, if a node is the i^{th} child of its parent, then the node is in quadrant i . The quadrant of the root is undefined.
- We define **subquadrant** i of a node to be the quadrant of child i of that node.
- The **conjugate** of quadrant i is defined to be the quadrant diagonally opposite. More formally, $conjugate(i) = (i + 2) \% 4$.
- For a node q in a quad tree, we define $T(q)$ as the set of nodes in the subtree rooted at q . More formally,

$$T(q) = \begin{cases} \emptyset, & \text{if } q \text{ is a frontier node} \\ q \cup \bigcup_{i=0,3} T(node.child(i)) & \text{otherwise.} \end{cases}$$

- For a node q in a quad tree, we define $S(q)$ to be the collection of elements in the subtree rooted at q . More formally,

$$S(q) = \begin{cases} \emptyset, & \text{if } q \text{ is a frontier node} \\ node.data \cup \bigcup_{i=0,3} S(node.child(i)) & \text{otherwise.} \end{cases}$$

Abstraction Function: A quad tree defines a collection of points consisting of the collection of elements in the nodes of the tree. More formally, if Q is a quad tree, then

$$AF(Q) \equiv S(Q.root).$$

Design Notes: One could require that the elements of a quad tree implement the *XYPoint* interface (Section 48.3). However, the *XYComparator* (Section 48.3) is used for generality. By defining a customized *XYComparator*, one can place elements of any data type into a quad tree, even if that data type had not been originally designed for use in a spatial collection.

Optimizations: The *remove* algorithm uses several heuristics to reduce the number of nodes that need to be rearranged during removal in order to maintain proper relative ordering of the elements within the tree. A simpler, but less efficient, alternative would be to reinsert all nodes in the subtree of the replacement node. The range search method *withinBounds* creates new *Box* objects (Section 48.5) defining the bounds and the current search space for each search. If the application requires many searches, one could initialize these boxes as instance variables and reuse them for each range search.

48.2 Representation Properties

We define the **INORDER** property to capture the desired relative positioning of elements in the subquadrants (subtrees) of each node. The **PARENT** property simply says that each parent/child relationship is kept consistent.

INORDER: For all quad tree nodes q and r , where $r \in T(q)$,

- $r \in T(q.child(0))$ iff $r.data$ is left of and below $q.data$,
- $r \in T(q.child(1))$ iff $r.data$ is right of and below $q.data$,
- $r \in T(q.child(2))$ iff $r.data$ is right of and above $q.data$, and
- $r \in T(q.child(3))$ iff $r.data$ is left of and above $q.data$.

PARENT: For all non-frontier nodes q in quad tree T , if $q.parent = null$, then $T.root = q$. Furthermore, $\forall i \in \{0, 1, 2, 3\}$, if $q.child(i)$ is not a frontier node, then $q.child(i).parent = q$.

48.3 Partitioning a Two-Dimensional Space

In the remainder of this chapter, we assume for simplicity that each element in the spatial collection is viewed as a two-dimensional point that implements the following *XYPoint* interface.

XY Point Interface

This interface is used in conjunction with the *DefaultXYComparator* to define the default comparator used to partition the space into four quadrants.

```
public interface XYPoint {

    public double getX();

    public double getY();

}
```

More generally, for a d -dimensional space, a *Point* interface, would replace the *getX* and *getY* methods, by a single *get(dim)* method that would take *dim*, the desired dimension. For example if we defined the x -coordinate as dimension 0, and the y -coordinate as dimension 1, then *getX* would be equivalent to *get(0)*, and *getY* would be equivalent to *get(1)*.

XYComparator

The *XYComparator* interface is used by the quad tree implementation to make decisions about its elements. In particular, the interface provides methods to get the x and y coordinates for an item, to compare items based on their x or y coordinates, and to determine which quadrant (0 = lower left, 1 = lower right, 2 = upper right, or 3 = upper left) an item belongs in, relative to a given origin. As a special case, the *quadrant* method returns the value 4 when the given points match in both x and y coordinates.

```
public interface XYComparator<T> extends Comparator<T> {
    public double getX(T item);
    public double getY(T item);
    public double compareX(T a, T b);
    public double compareY(T a, T b);
    public int quadrant(T origin, T item);           //0, 1, 2, 3, or 4 (match)
}
```

Default XY Comparator

```

public class DefaultXYComparator<E extends XYPoint> implements XYComparator<E> {

    public double compareX(E a, E b) {return a.getX() - b.getX();}
    public double compareY(E a, E b) {return a.getY() - b.getY();}

    public double getX(E item) { return item.getX();}
    public double getY(E item) { return item.getY();}

    public int compare(E a, E b) {                                     //uses X comparisons by default
        double d = compareX(a, b);
        if (d < 0)
            return -1;
        else if (d == 0)
            return 0;
        else
            return 1;
    }
}

```

The *quadrant* method takes *origin*, the element that defines the origin, and *target*, the target element. It returns the quadrant (with respect to *origin*) that contains the target.

```

public int quadrant(E origin, E target) {
    if (getX(origin) == getX(target) && getY(origin) == getY(target))
        return 4;
    boolean toLeft = compareX(target, origin) ≤ 0;
    boolean below = compareY(target, origin) ≤ 0;
    if (toLeft)
        return below ? 0 : 3;
    else
        return below ? 1 : 2;
}
}

```

48.4 QTreeNode Inner Class

TreeNode

└ *QTreeNode*

Each quad tree node contains one element. The array of children, indexed by quadrant number, contains references to its four child nodes. Each node also contains a parent reference, which is *null* if the node is the root of the tree.

```

E data;
Object[] children = new Object[4];
QTreeNode parent;

```

The QTreeNode constructor takes as its parameter *data*, the element to be stored in the node. It initially sets all children of the new node to be the frontier node.

```

protected QTreeNode(E data) {
    this.data = data;
    for (int i = 0; i < 4; i++)
        children[i] = FRONTIER;
}

```

The *isFrontier* method returns *true* if this node is the frontier node.

```

protected boolean isFrontier() {
    return this == FRONTIER;
}

```

Since each quad tree node holds one element, the *size* method returns 1 for non-frontier nodes, or 0 if this is the frontier node.

```

protected int size() {
    return isFrontier()? 0 : 1;
}

```

Similarly, the *capacity* method returns 1 for non-frontier nodes, or 0 if this is the frontier node.

```

protected int capacity() {
    return isFrontier()? 0 : 1;
}

```

The *data* method takes *index*, an ignored parameter, and returns the element held in this node. The parameter is provided since tree nodes may, in general, contain multiple values. However, quad tree nodes contain only one value.

```

protected E data(int index) {
    return data;
}

```

The *child* method takes *index*, the index of the desired child (0-3), and returns the child quad tree node with that index. Note that the return value may be the frontier node. Since there are exactly four quadrants around each point, with four corresponding subtrees, the method requires that the index is in the range 0 to 3, inclusive.

```

protected QTreeNode child(int index) {
    return (QTreeNode) children[index];
}

```

The *indexOf* method takes as its parameter *child*, a quad tree node. If the given child is one of the four children of this node, then the method returns the index of the given child within this node, or -1 if the given node is not a direct child of this node.

```

int indexOf(QTreeNode child) {
    for (int q=0; q<4; q++)
        if (child(q) == child)
            return q;
    return -1;
}

```

The *setChild* method takes as its parameters *index*, a child index, and *child*, a quad tree node. The effect of the method is to make the given node a child of this node at the given index. This method requires that the index is in the range 0 to 3, inclusive, and that this node is not in the subtree of the given child. The child's parent reference is set in order to preserve the PARENT property. Note, however that if there had already been a child of this node at the given index, this method does not reset the parent reference of that former child.

```
protected void setChild(int index, QTNode child) {
    children[index] = child;
    if (child != null)
        child.parent = this;
}
```

The *resetParent* method sets the parent of a quad tree node to null. To preserve the PARENT property, it must also reset the appropriate child reference of its former parent, if any.

```
public void resetParent() {
    if (parent != null) {
        parent.children[parent.indexOf(this)] = FRONTIER;
        parent = null;
    }
}
```

When an element is removed from a quad tree, its node must be replaced by another node to preserve the structure of the tree. The *replaceBy* method takes *replacement*, a replacement quad tree node, and repositions that node in place of this one. To preserve the INORDER property, it is expected that the replacement node is in the subtree of this node, and additional restructuring of the tree may be required. The replacement is accomplished by setting the parent of the replacement node to be the parent of this node, as well as setting the children of the replacement node to be the children of this node, provided that the replacement is not the frontier node. An alternative to copying all of these references would be to copy the data of the other node into this one. However, keeping the data with its original node sets the stage for a possible tracked version of this data structure.

```
public void replaceBy(QTNode replacement) {
    if (parent == null) {
        root = replacement;
        replacement.parent = null;
    } else {
        parent.setChild(parent.indexOf(this), replacement);
    }
    if (replacement != FRONTIER) {
        for (int q = 0; q < 4; q++)
            replacement.setChild(q, child(q));
    }
    parent = null;
}
```

At one point in the remove algorithm, it is necessary to find the deepest node in a subtree along a particular quadrant line. The *deepestInSubtree* method takes *quadrant*, the desired search quadrant

and drills down into the subtree of this node as far as possible, until it returns the deepest non-frontier node found by following the chain of children along that quadrant index, or this node itself if it happens to be the frontier node.

```
protected QTreeNode deepestInSubtree(int quadrant) {
    if (isFrontier())
        return this;
    QTreeNode ptr = this;
    while (!ptr.child(quadrant).isFrontier())
        ptr = ptr.child(quadrant);
    return ptr;
}
```

To support traversing the data structure, the *traverseForVisitor* method takes a *v*, a visitor, and recursively traverses the data structure, allowing the visitor to see each non-frontier node. Since the quad tree does not provide its own custom iterator, this method provides the traversal algorithm for the visiting iterator. (See the quad tree *iterator* method.)

```
public void traverseForVisitor(Visitor<? super E> v) throws Exception {
    if (!isFrontier()) {
        v.visit(data);
        for (int q = 0; q < 4; q++)
            child(q).traverseForVisitor(v);
    }
}
```

48.5 Box Inner Class

The Box inner class is used for range searching within a quad tree. It is also used within the remove algorithm to keep track of the “danger zone” containing points that need to be reinserted as the result of replacing the removed element by an element from its subtree.

A box defines a rectangle with minimum and maximum *x* and *y* values.

```
double minx, miny, maxx, maxy;
```

The most general constructor takes as parameters *minx*, the minimum *x* coordinate, *miny*, the minimum *y* coordinate, *maxx*, the maximum *x* coordinate, and *maxy*, the maximum *y* coordinate, and creates a box with those boundary values.

```
Box(double minx, double miny, double maxx, double maxy) {
    set(minx, miny, maxx, maxy);
}
```

Several constructors are provided for convenience. The no-argument constructor creates a box (0,0,0,0).

```
Box() { }
```

The constructor that takes *a*, an arbitrary quad tree node, and *b*, a second quad tree node creates a box whose boundary is defined by the points corresponding to the elements in the given nodes.

```
Box(QTNode a, QTNode b) {
    setCorners(a.data, b.data);
}
```

The constructor that takes *a*, an arbitrary quad tree element, and *b*, a second quad tree element creates a box whose boundary is defined by the points corresponding to the given elements.

```
Box(E a, E b) {
    setCorners(a, b);
}
```

The *set* method takes as parameters *minx*, the minimum *x* coordinate, *miny*, the minimum *y* coordinate, *maxx*, the maximum *x* coordinate, and *maxy*, the maximum *y* coordinate, and sets box to those boundary values. The method returns this box for convenience.

```
Box set(double minx, double miny, double maxx, double maxy) {
    this.minx = minx;
    this.miny = miny;
    this.maxx = maxx;
    this.maxy = maxy;
    return this;
}
```

The *setCorners* method takes *a*, an arbitrary quad tree element and *b*, a second quad tree element sets the box to the boundary defined by the points corresponding to the given elements. The *x* and *y* values are extracted from each element using the quad tree's comparator, and then minimum and maximum *x* and *y* values are determined. Therefore, this method can be called with arbitrary opposite corners of the desired bounding box.

```
void setCorners(E a, E b) {
    double ax = comp.getX(a), ay = comp.getY(a);
    double bx = comp.getX(b), by = comp.getY(b);
    double minx = Math.min(ax, bx), maxx = Math.max(ax, bx);
    double miny = Math.min(ay, by), maxy = Math.max(ay, by);
    set(minx, miny, maxx, maxy);
}
```

The *intersects* method takes as its parameter *b*, another box, and returns *true* if and only if the given box has a non-empty intersection with this box.

```
final boolean intersects(Box b) {
    return b.contains(minx, miny) ||
        contains(b.minx, b.miny) || contains(b.minx, b.maxy) ||
        contains(b.maxx, b.miny) || contains(b.maxx, b.maxy);
}
```

The *contains* method that takes as its parameter *e*, an element, returns *true* if and only if the (*x*, *y*) point corresponding to the given element is within, or on the boundary of, this box.

```
final boolean contains(E e) {
    return contains(comp.getX(e), comp.getY(e));
}
```

The *contains* method that takes as parameters *x*, an *x* coordinate and *y*, a *y* coordinate, returns *true* if and only if the given (*x*, *y*) point is within, or on the boundary of, this box.


```

final boolean contains(double x, double y) {
    return (minx ≤ x && x ≤ maxx) && (miny ≤ y && y ≤ maxy);
}

```

The *inRange* method that takes as parameters *x*, an *x* coordinate and *y*, a *y* coordinate, returns *true* if and only if either the *x* coordinate is within the (inclusive) range of the minimum and maximum *x* coordinates of the box, or the *y* coordinate is within the (inclusive) range of the minimum and maximum *y* coordinates of the box. This is used in the remove algorithm to determine whether a given node in the tree must be reinserted. Note that a true value does *not* necessarily imply that the point (*x*, *y*) is within, or on the boundary of, this box. Instead, think of the point being within the infinite “cross” whose vertical and horizontal bands intersect exactly on this box. Examples of such shapes are shown in [Figure 48.3](#) on page 775.

```

boolean inRange(double x, double y) {
    return (minx ≤ x && x ≤ maxx) || (miny ≤ y && y ≤ maxy);
}

```

The *inRange* method that takes as its parameter *node*, an arbitrary quad tree node, returns *true* if and only if the node is not a frontier node and the (*x*, *y*) point corresponding to the node is within the infinite “cross” whose vertical and horizontal bands are defined by the *x* and *y* ranges of this box.

```

boolean inRange(QTNode node) {
    if (node.isFrontier())
        return false;
    double cx = comp.getX(node.data);
    double cy = comp.getY(node.data);
    return inRange(cx, cy);
}

```

48.6 Quad Tree Methods

In this section, we present the methods for the *QuadTree* class.

48.6.1 Constructors and Factory Methods

The default constructor uses a default comparator, under the assumption that type *E* implements the *XYPoint* interface. If type *E* does not implement *XYPoint*, then an *XYComparator* for *E* should be provided, using the other constructor. Otherwise, a *ClassCastException* will occur when the second element is added to the quad tree.

```

public QuadTree() {
    this(new DefaultXYComparator());
}

```

The following constructor takes as its parameter *comp*, an *XYComparator* for comparing the elements of this collection. A variable named *comp* is inherited from the *AbstractCollection* class and initialized in the super constructor. However, to avoid casting throughout this class, a second *comp* variable is declared in this class with a more specific type and initialized here.

```

public QuadTree(XYComparator<? super E> comp) {
    super(comp);
    this.comp = (XYComparator<? super E>) comp;
}

```

The usual factory method, *createTreeNode* is declared to create tree nodes of the appropriate type.

```

protected QTreeNode createTreeNode(E data) {
    return new QTreeNode(data);
}

```

48.6.2 Representation Accessors

Several convenience methods are defined to express common relationships among quadrants. The method *conjugate* for *q*, a given quadrant, returns the opposite quadrant.

```

static final int conjugate(int q) {
    return (q+2) % 4;
}

```

The method *leftNeighbor* for *q*, a given quadrant, returns the adjacent quadrant in the clockwise direction.

```

static final int leftNeighbor(int q) {
    return (q+3) % 4;
}

```

The method *rightNeighbor* for *q*, a given quadrant, returns the adjacent quadrant in the counter-clockwise direction.

```

static final int rightNeighbor(int q) {
    return (q+1) % 4;
}

```

48.6.3 Algorithmic Accessors

The *max* method takes as its parameter a *dimension*, where 0 is the *x* dimension and 1 is the *y* dimension, and returns a maximum element along the given dimension.

```

public E max(int dimension) {
    if (dimension < 0 || dimension > 1)
        throw new IllegalArgumentException("dimension 0 or 1 required");
    if (isEmpty())
        throw new NoSuchElementException();
    return max(root.data, root, dimension);
}

```

The auxiliary *max* method takes as parameters *candidate*, the maximum found so far, *subtree*, the root of the subtree in which to search, and *dimension*, where 0 is the *x* dimension and 1 is the *y* dimension. It returns a maximum element among the given candidate and the elements in the given subtree, along the given dimension of comparison.

```

private E max(E candidate, QTNode subtree, int dimension) {
    if (!subtree.isFrontier()) {
        candidate = max(subtree.data, subtree.child(2), dimension);
        if (dimension == 0) {                                     //x dimension
            E b = max(subtree.data, subtree.child(1), dimension);
            if (comp.compareX(b, candidate) > 0)
                candidate = b;
        } else {                                               //y dimension
            E b = max(subtree.data, subtree.child(3), dimension);
            if (comp.compareY(b, candidate) > 0)
                candidate = b;
        }
    }
    return candidate;
}

```

Correctness Highlights: If the subtree is a frontier node, then the subtree contains no elements, so the maximum element is simply the provided candidate. Otherwise, two quadrants must be searched, depending on the dimension of comparison: if the x dimension is used, then quadrants 1 and 2 must be searched since, by INORDER, together they contain all points in the subtree with x coordinates greater than the x coordinate for the root of the subtree. Similarly, if the y dimension is used, then quadrants 2 and 3 must be searched. In either case, we start by taking as a new candidate the maximum of this node's data and the elements in quadrant 2. Then the maximum in the other quadrant (1 or 3, for dimension x or y) is compared to the candidate and the maximum of the two is returned. ■

The *min* method takes as its parameter a *dimension*, where 0 is the x dimension and 1 is the y dimension, and returns a minimum element along the given dimension.

```

public E min(int dimension) {
    if (dimension < 0 || dimension > 1)
        throw new IllegalArgumentException("dimension 0 or 1 required");
    if (isEmpty())
        throw new NoSuchElementException();
    return min(root.data, root, dimension);
}

```

The auxiliary *min* method takes as parameters *candidate*, the minimum found so far, *subtree*, the root of the subtree in which to search, and *dimension*, where 0 is the x dimension and 1 is the y dimension. It returns a minimum element among the given candidate and the elements in the given subtree, along the given dimension of comparison.

```

private E min(E candidate, QTNode subtree, int dimension) {
    if (!subtree.isFrontier()) {
        candidate = min(subtree.data, subtree.child(0), dimension);
        if (dimension == 0) {                                     //x dimension
            E b = min(subtree.data, subtree.child(3), dimension);
            if (comp.compareX(b, candidate) < 0)
                candidate = b;
        } else {                                               //y dimension
            E b = min(subtree.data, subtree.child(1), dimension);

```

```

        if (comp.compareY(b, candidate) < 0)
            candidate = b;
    }
}
return candidate;
}

```

Correctness Highlights: If the subtree is a frontier node, then the subtree contains no elements, so the minimum element is simply the provided candidate. Otherwise, two quadrants must be searched, depending on the dimension of comparison: if the x dimension is used, then quadrants 0 and 3 must be searched since, by INORDER, together they contain all points in the subtree with x coordinates less than the x coordinate for the root of the subtree. Similarly, if the y dimension is used, then quadrants 0 and 1 must be searched. In either case, we start by taking as a new candidate the minimum of this node's data and the elements in quadrant 0. Then the minimum in the other quadrant (3 or 1, for dimension x or y , respectively) is compared to the candidate and the minimum of the two is returned. ■

The auxiliary recursive method *withinBounds* takes as its parameters *node*, the root of a given subtree, *bounds*, the bounds within which the desired elements lie, *space*, the extent of the search space covered by the subtree, and *elements*, a collection of elements that have been discovered within the bounds so far. The method adds, to the given collection of elements, all elements in the given subtree that lie within the given bounds. The *space* parameter is used to avoid searching subtrees whose extents have an empty intersection with the desired bounds. The method requires that all elements of the given subtree lie within the given space, which may or may not intersect the desired bounding box.

```

void withinBounds(QTNode node, Box bounds, Box space,
                  Collection<? super E> elements) {
    if (!node.isFrontier() && bounds.intersects(space)) {
        if (bounds.contains(node.data))
            elements.add(node.data);
        double xMin = space.minx, yMin = space.miny;
        double xMax = space.maxx, yMax = space.maxy;
        double x = comp.getX(node.data), y = comp.getY(node.data);
        withinBounds(node.child(0), bounds, space.set(xMin, yMin, x, y), elements);
        withinBounds(node.child(1), bounds, space.set(x, yMin, xMax, y), elements);
        withinBounds(node.child(2), bounds, space.set(x, y, xMax, yMax), elements);
        withinBounds(node.child(3), bounds, space.set(xMin, y, x, yMax), elements);
    }
}

```

Correctness Highlights: The given subtree can contain nodes within the given bounds only if the given node is not a frontier node and the bounds intersects the search space defined by the node's subtree. Provided these conditions are met, the method first examines the element e contained in the root of the subtree to determine if its corresponding point p lies within the desired bounds. If so, e is added to the resulting collection. Four recursive calls handle the subtrees. For each recursive call, the search space is narrowed according to p . For example, since quadrant 0 contains elements corresponding to points only to the left and below p , the space passed to the recursive call for quadrant 0 is bounded to the right and above by $p.x$ and $p.y$, respectively. ■

The *withinBounds* method takes as its parameters *cornerA*, an element defining one corner of a bounding box and *cornerB*, an element defining the opposite corner of the bounding box. The method returns a collection of all elements in this quad tree that lie within, or on the boundary of, this bounding box, based on the XYComparator of this quad tree.

```
public Collection<E> withinBounds(E cornerA, E cornerB) {
    Collection<E> elements = new DoublyLinkedList<E>();
    withinBounds(cornerA, cornerB, elements);
    return elements;
}
```

A second public *withinBounds* method takes as its parameters *cornerA*, an element defining one corner of a bounding box, *cornerB*, an element defining the opposite corner of the bounding box, and *elements*, the collection in which to place the elements that lie within, or on the boundary of, the bounding box. Using this method avoids the creation of a new collection on each call. Furthermore, the method provides the flexibility to add the desired elements to any type of collection, including another spatial collection.

```
public void withinBounds(E cornerA, E cornerB, Collection<? super E> elements) {
    if (elements == this)
        throw new IllegalArgumentException(“a different collection must be provided”);
    Box bounds = new Box(cornerA, cornerB);
    Box searchSpace = new Box(NEGATIVE_INFINITY, NEGATIVE_INFINITY,
                             POSITIVE_INFINITY, POSITIVE_INFINITY);
    withinBounds(root, bounds, searchSpace, elements);
}
```

Correctness Highlights: Correctness follows from the auxiliary recursive *withinBounds* method, which is called with the root of the quad tree, the given bounding box, a search space that is infinite in all directions, and an empty collection to which the desired elements are added. ■

The method *find* takes as its parameter *target*, any value of type E and returns the tree node containing an element equal to the target, or the frontier node if no matching element is found. If the frontier node is returned, then its parent will have been set to the last non-frontier node on the search path.

```
protected TreeNode find(E target) {
    QTNode ptr = root;
    FRONTIER.parent = null;
    while (!ptr.isFrontier()) {
        int quadrant = comp.quadrant(ptr.data, target);
        if (quadrant == 4) //found a match
            return ptr;
        else {
            FRONTIER.parent = ptr;
            ptr = ptr.child(quadrant);
        }
    }
    return ptr;
}
```

Correctness Highlights: The search begins at the root of the quad tree. By INORDER, we know that at each node, either we find a match at that node or we can use the x and y coordinates of the node's element and the target to determine a single quadrant in which the search should proceed. This decision is made by the comparator, which returns 4 if a match is found. At each iteration, the parent of the frontier node is set to the current node before proceeding down the tree to the next level. In this way, when a frontier node is reached in the search path, its parent already will have been set to the last non-frontier node on the search path. ■

The method *findInsertPosition* takes as its parameters *subtree*, a quad tree node at the root of a subtree, and *element*, an element for insertion into the quad tree. The method does not modify the structure of the tree, but simply returns the frontier node at the insert position of the target. That is, the frontier node's parent is set to the quad tree node that should become the parent of the target.

```
protected QTreeNode findInsertPosition(QTreeNode subtree, E element) {
    FRONTIER.parent = null;
    while (!subtree.isFrontier()) {
        FRONTIER.parent = subtree;
        subtree = subtree.child(comp.quadrant(subtree.data, element)%4);
    }
    return subtree;
}
```

Correctness Highlights: The correctness argument is similar to that of the *find* method, except that traversal continues to quadrant 0 when there is a matching element in the tree. ■

48.6.4 Content Mutators

Methods to Perform Insertion

The *insert* method takes an *element*, the element to be inserted into the tree. It delegates the insertion to the *insertNode* method, to which it passes a new node containing the element, as well as the root of the quad tree into which the new node should be inserted.

```
protected QTreeNode insert(E element) {
    return insertNode(new QTreeNode(element), root);
}
```

The *insertNode* method takes *node*, a quad tree node and *subtree*, the root of the subtree into which the node should be inserted. The method places the element in the subtree at the correct location.

```
protected QTreeNode insertNode(QTreeNode node, QTreeNode subtree) {
    E element = node.data;
    QTreeNode parent = findInsertPosition(subtree, element).parent;
    if (parent == null)
        root = node;
    else {
```

```

        node.parent = parent;
        int quadrant = comp.quadrant(parent.data, element)%4;
        parent.children[quadrant] = node;
    }
    return node;
}

```

Correctness Highlights: Correctness relies on that of *findInsertPosition* which is used to determine the placement for the node. The remaining decision is whether or not the new node will become the root of the tree, which occurs when the frontier node returned by *findInsertPosition* has a null parent. Otherwise, the frontier node's parent is the proper parent of the node being inserted, and that parent's child pointer is set for the appropriate quadrant according to the comparator, with a matching element going to quadrant 0. ■

Methods to Perform Deletion

The public *remove* method takes *element*, an element in the collection and removes the given element from the collection. It calls the inherited *remove* method and increments the version count to invalidate any iterators. The inherited method finds the tree node containing the given element, and then calls the *remove* method in this class that takes a tree node as a parameter to do the actual work. This method returns *true* if and only if the given element was found in the collection.

```

public boolean remove(E element) {
    if (super.remove(element)) {
        version.increment();
        return true;
    }
    return false;
}

```

The algorithm to remove a node is complex. We describe the algorithm in general, but occasionally support our explanation by referring to the problem of removing the root (100,150) from the quad tree shown at the top of [Figure 48.2](#). Throughout the discussion, we use the term *deleted node* to refer to the node containing the element to be deleted from the quad tree, even though the node is not technically deleted from the tree until the execution has completed. Also, where clear from context, we sometimes refer to a node as if it were the point corresponding to the element contained in that node, as determined by the comparator.

As in a binary search tree (Chapter 32), removal of a non-leaf node requires finding a *replacement node*, which will take the place of the deleted node in the tree and adopt the children of the deleted node. However, unlike in a binary search tree, it is not possible in general to find a replacement quad tree node that can be moved into the position of the deleted node without violating the INORDER property. Consequently, some other nodes may need to be moved as well. The reason why a “perfect” replacement node is not generally possible is the direct result of working in two dimensions simultaneously: a node that is the predecessor in one dimension may not be the predecessor in the other dimension, so some nodes that were in one subquadrant of the deleted node may need to be shifted to a different subquadrant of the replacement node. A brute force algorithm might choose the replacement node arbitrarily and then reinsert all of the other nodes in the subtree of the deleted node. However, the deletion algorithm presented here [130] attempts to limit reinsertions in two ways. First, the replacement node is chosen carefully to limit the need for reinsertions. Second,

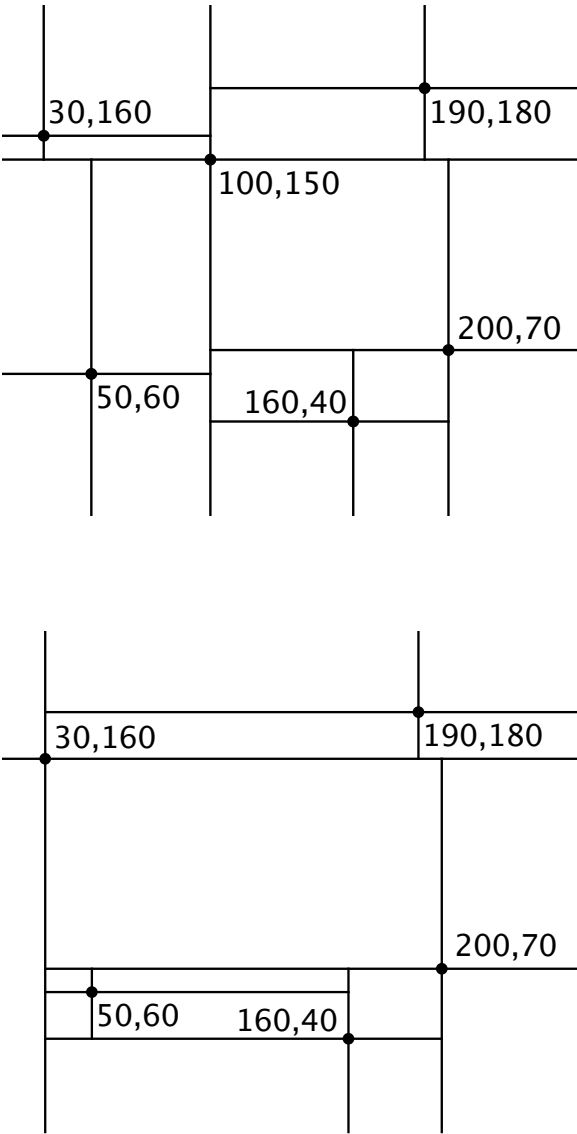


Figure 48.2
Before and after deleting the root node (100, 150) from a quad tree. Following deletion, the new root is (30, 160).

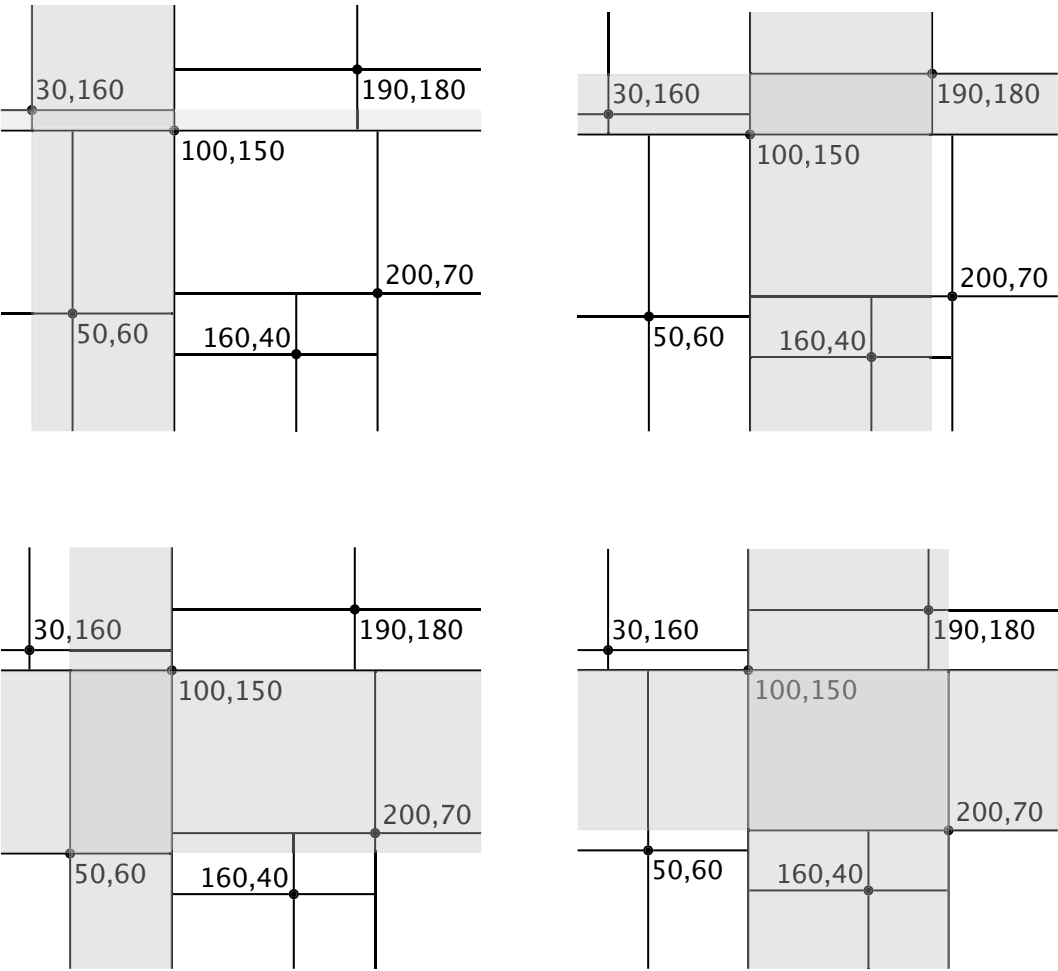


Figure 48.3
Shaded danger zones for candidate replacement nodes in each of the four subtrees of the root. The candidate nodes, counter clockwise from the lower left, are (50,60), (200,70), (190,180), and (30,160). Note, however, that candidates need not be the immediate children of the node to be deleted.

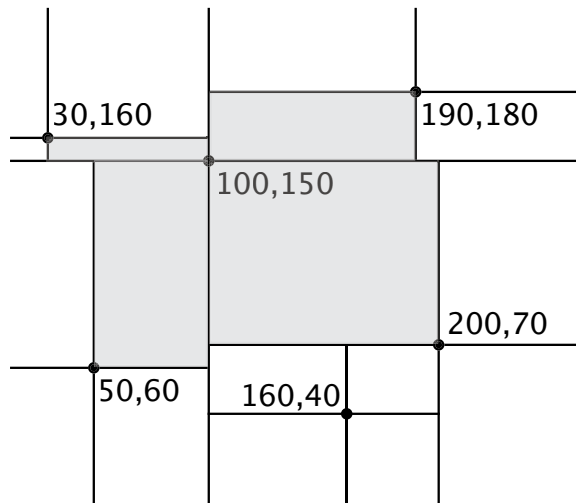


Figure 48.4

Rectangles whose perimeters correspond to the L1 metric for each candidate replacement node.

the algorithm exploits the structure of the tree to examine only those portions of the deleted node's subtree that may contain nodes requiring reinsertion.

The first step in removing a node is searching its subtree for the replacement node that will take the deleted node's position in the quad tree. Each possible replacement induces a "danger zone," a region of space containing those points whose nodes must be reinserted into the tree when the replacement node is moved into the position of the deleted node. For example, Figure 48.3 shows, as shaded regions, cross-shaped danger zones for choices of four different possible replacement nodes in deleting the root node (100, 50). Ideally, we would like to choose a replacement node whose danger zone contains the fewest nodes. However, it would take too much time to search the entire tree for the node whose danger zone includes the fewest nodes. Instead, we employ a series of three efficient heuristics to find a replacement whose danger zone is small, and therefore unlikely to contain many other nodes. The heuristics are (1) identify a candidate from each subquadrant, (2) filter the candidates, and finally (3) break ties if more than one candidate remains after filtering. To identify a candidate from each subquadrant q of the node, we find the node "nearest" to the deleted node by searching as far as possible down that subtree toward the node in the direction *conjugate*(q). To filter the candidates, we determine if any of these four candidates contains no other candidates in its danger zone. This is equivalent to finding candidates that are closer to the deleted node, in both x and y , than the candidates in the neighboring quadrants on the same side of the x and y axis, respectively. If any such candidates are found, the set of candidates is filtered to include only those (1 or 2) candidates. (In the example of Figure 48.3, there are no such candidates, since each of the four contains at least one other candidate in its danger zone, so none would be filtered out.) After identifying and possibly filtering the candidates, the third heuristic breaks ties among the remaining candidates by choosing the one whose Manhattan distance (total of x and y distance) from the deleted node is least. This Manhattan distance, known here as the *L1 metric*, essentially compares the shaded rectangles in Figure 48.4 for each candidate to choose the one whose rectangle has the smallest perimeter. This is reasonable since, assuming a square non-infinite search space, this perimeter is proportional to the area of the danger zone induced by that candidate.

Heuristic 1: The method *identifyReplacementCandidate* takes as its parameter *node*, the node to be deleted, and selects one candidate from each of the deleted node's subquadrants. For each subquadrant q , the method traverses down the subtree in the direction *conjugate*(q). For example,

when traversing the upper left subquadrant (child 3 of the node), traversal proceeds back toward the node, towards the lower right by continually following child 1 of that subquadrant, which is the conjugate of 3. This results in a node from each quadrant that is, in some sense, “closest” to the given node. The frontier node is taken as the candidate for each empty subquadrant.

As each candidate is identified, three arrays (indexed by quadrant number) are filled with information about the candidates. The *candidates* array contains a reference to the candidate identified from each quadrant. The *xDistance* and *yDistance* arrays contain the distances in *x* and *y* between the candidate and the node to be deleted. (Frontier nodes are considered infinitely far away from the node.) These arrays are kept as instance variables in the quad tree for use by other methods in the remove algorithm, and so that new arrays do not need to be allocated each time remove is called.

```
void identifyReplacementCandidates(QTNode node) {
    int candidateCount = 0;
    for (int q = 3; q ≥ 0; q--) {
        QTNode deepest = node.child(q).deepestInSubtree(conjugate(q));
        candidates[q] = deepest;
        if (deepest.isFrontier()) {
            xDistance[q] = yDistance[q] = Double.POSITIVE_INFINITY;
        } else {
            xDistance[q] = Math.abs(comp.getX(node.data) - comp.getX(deepest.data));
            yDistance[q] = Math.abs(comp.getY(node.data) - comp.getY(deepest.data));
            candidateCount++;
        }
    }
    if (candidateCount > 1)
        filterCandidates();
}
```

Correctness Highlights: Correct identification of each candidate follows from the INORDER property and the correctness of the *deepestInSubtree* and *conjugate* methods. The distances for each quadrant’s candidate are computed directly from values provided by the comparator. Filtering of the candidates (Heuristic 2) is required only if more than one non-frontier candidate is found. ■

Heuristic 2: The method *filterCandidates* examines the arrays filled by the *identifyReplacementCandidates* method to determine if any of the candidates contain no other candidates in its danger zone. Although these special candidates still may contain other nodes in their danger zones, we know that at least they contain fewer than other candidates. If any such special candidates are found, the other (non-special) candidates are filtered out by removing them from the *candidates* array.

```
void filterCandidates() {
    boolean found = false; //true if a special candidate has been found
    int xOffset = 3; //x comparisons between quadrants 0/3, 1/2, 2/1, and 3/0
    int yOffset = 1; //y comparisons between quadrants 0/1, 1/0, 2/3, and 3/2
    for (int q = 0; q < 4; q++) {
        if (xDistance[q] < xDistance[(q+xOffset)%4] &&
            yDistance[q] < yDistance[(q+yOffset)%4]) {
            if (!found) { //the first special candidate
                found = true;
                for (int i = 0; i < q; i++) //filter out earlier candidates

```

```

        candidates[i] = null;
    }
} else if (found) {
    candidates[q] = null;
}
int temp = xOffset; xOffset = yOffset; yOffset = temp;
}
}

```

Correctness Highlights: For each subquadrant, it is necessary to compare that subquadrant's candidates with the candidates in the neighboring subquadrants. When comparing to a neighbor that is left or right of a quadrant, a special candidate must have a y distance to the deleted node that is less than its neighboring candidate's distance. When comparing to a neighbor that is in the quadrant above or below, a special candidate must have an x distance to the deleted node that is less than its neighboring candidate's distance. To implement this, we begin at quadrant 0 with an $xOffset$ of 3, so that the candidate in quadrant 0 is compared with the candidate in quadrant 3 in terms of its x distance from the deleted node. Similarly, we begin at quadrant 0 with a $yOffset$ of 1, so that the candidate in quadrant 0 is compared with the candidate in quadrant 1 in terms of its y distance from the deleted node. As we proceed to each next quadrant, the values of $xOffset$ and $yOffset$ are swapped, so that each quadrant's candidates are compared as follows. The candidates in quadrants 0, 1, 2, and 3 are compared to those in quadrants 3, 2, 1, and 0, respectively, in terms of x distance. Similarly, the candidates in quadrants 0, 1, 2, and 3 are compared to those in quadrants 1, 0, 3, and 2, respectively, in terms of y distance. When the first special candidate is found (if any), all previously considered candidates are removed from the *candidates* array, and those candidates considered in the future are removed if they are found not to be special. ■

Heuristic 3: After candidates have been identified and filtered, the final heuristic compares the (remaining) candidates using the $L1$ metric, meaning the sum of the x and y distances from the candidate to the deleted node. The selected candidate is the one whose $L1$ metric is least.

The method *chooseReplacementNode* takes as its parameter *node*, the node to be deleted, and returns the number of the subquadrant containing the replacement node. The node itself can then be extracted from the *candidates* array at the index of that subquadrant.

```

int chooseReplacementNode(QTNode node) {
    identifyReplacementCandidates(node);
    int winningQuadrant = 0;
    double distance = java.lang.Double.MAX_VALUE;
    for (int q = 0; q < 4; q++) {
        if (candidates[q] != null) {
            double L1 = xDistance[q] + yDistance[q];
            if (L1 < distance) {
                distance = L1;
                winningQuadrant = q;
            }
        }
    }
    return winningQuadrant;
}

```

Correctness Highlights: The first two steps of choosing a replacement (identification and filtering) are delegated to the *identifyReplacementCandidates* method, which calls *filterCandidates* if necessary. The last heuristic is implemented within the method itself, as a straightforward comparison of the sums of the *x* and *y* distances recorded for each candidate. ■

In the remainder of the discussion of the remove algorithm, we refer to the selected replacement node as the *winner* and its subquadrant relative to the deleted node as the *winning quadrant*. The *remove* method takes as its parameter *toDelete*, the quad tree node to be deleted. It removes the node from the tree and restores the INORDER property. The method begins by choosing a replacement node using the methods described above. This winner and the deleted node together define the danger zone within which all nodes must be reinserted for proper placement in the tree. The winner cannot be moved immediately because we need to retain its subtrees for reinsertion. However, before reinserting any nodes, we must (temporarily) copy the winner's data into the node that will be deleted, so that comparisons will be made correctly as nodes are reinserted. We then splice the winner (and its subtree) out of the tree before examining the subtree of the deleted node for possible node reinsertions. This examination does not necessarily reinsert all of the nodes in the subtree, but only examines those parts of the subtree that may fall within the danger zone. All nodes in the winner's subtree are then reinserted. Finally, the deleted node is replaced by the winner.

```
protected void remove(TreeNode toDelete) {
    QTreeNode node = (QTreeNode) toDelete;
    if (node.isFrontier()) return;

    int winningQuadrant = chooseReplacementNode(node);
    QTreeNode winner = (QTreeNode) candidates[winningQuadrant];

    if (!winner.isFrontier()) {
        dangerZone.setCorners(winner.data, node.data);
        node.data = winner.data;
        winner.resetParent();
        adjustSubtree(node, winner, winningQuadrant);
        for (int q = 0; q < 4; q++)
            reinsertEntireSubtree(winner.child(q), node);
    }
    node.replaceBy(winner);
}
```

When a node in the danger zone is encountered, it is necessary to reinsert its entire subtree. To save time, it can often be determined that the reinsertion can begin at a node other than the root of the entire quad tree. Therefore, the method *reinsertEntireSubtree* takes two parameters: *subtreeRoot*, the root of the subtree to be reinserted, and *destinationRoot*, the root of the subtree into which the nodes should be reinserted. The method resets the parent of each node in the subtree (which includes resetting the appropriate child pointer of that parent) and then reinserts the node into the given subtree. The method requires that the *destinationRoot* is not in the subtree of the *subtreeRoot*.

```
void reinsertEntireSubtree(QTreeNode subtreeRoot, QTreeNode destinationRoot) {
    if (subtreeRoot != FRONTIER) {
        subtreeRoot.resetParent();
        insertNode(subtreeRoot, destinationRoot);
        for (int q = 0; q < 4; q++) {
            reinsertEntireSubtree(subtreeRoot.child(q), destinationRoot);
        }
    }
}
```

```

    }
  }
}

```

Correctness Highlights: The algorithm is a preorder traversal of the subtree, inserting each node as it is visited. Although the children of each node are not removed before it is reinserted into the tree, we know by INORDER that no node in this subtree can ever change subquadrants of this node on reinsertion. Therefore, the fact that the children remain in place until they are encountered in the algorithm does not affect any of the other reinsertions. ■

To adjust the subtree of the deleted node, certain quadrants below that node must be examined for possible reinsertion of nodes. Clearly, the quadrant that is the conjugate of the winning quadrant need not be examined, since all of its nodes are clearly on the same side of the replacement as they were of the deleted node. The two quadrants adjacent to the winning quadrant must be examined since they may contain nodes in the danger zone. Finally, within the winning quadrant itself, we must consider each node p along the path from the deleted node toward the winning node (in the direction of the conjugate quadrant of the winning quadrant). For each node p on the path, there are two possible cases: (1) p is in the danger zone, in which case we must reinsert its entire subtree, or (2) p is not in the danger zone, in which case we must consider its two quadrants on either side of the path toward the winning node (i.e., the two quadrants whose indices are neither the winning quadrant index nor its conjugate).

```

void adjustSubtree(QTNode deletedNode, QTNode winner, int winningQuadrant) {
    adjustSubquadrant(deletedNode, leftNeighbor(winningQuadrant),
        leftNeighbor(winningQuadrant), winningQuadrant, root);
    adjustSubquadrant(deletedNode, rightNeighbor(winningQuadrant),
        rightNeighbor(winningQuadrant), winningQuadrant, root);

    QTNode p = deletedNode.child(winningQuadrant);
    int conjugate = conjugate(winningQuadrant);
    while (!p.isFrontier()) {
        if (dangerZone.inRange(p)) {
            reinsertEntireSubtree(p, deletedNode);
            break;
        } else {
            adjustSubquadrant(p, winningQuadrant,
                leftNeighbor(winningQuadrant), winningQuadrant, deletedNode);
            adjustSubquadrant(p, winningQuadrant,
                rightNeighbor(winningQuadrant), winningQuadrant, deletedNode);
            p = p.child(conjugate);
        }
    }
}

```

Correctness Highlights: Let c be the immediate child of the deleted node in the winning quadrant q . By INORDER, we know that all nodes in the subtree q of c need not be examined, since they have the same relationship to the replacement node as the deleted node. The same argument applies for the nodes along the path from c to the replacement node. Consequently, only the quadrants on either side of the path need to be examined, unless a node on the path is

discovered to be in the danger zone, in which case its entire subtree is reinserted. The remainder of the correctness argument follows from *reinsertEntireSubtree*, as well as *adjustSubquadrant* (below) which examines the specific subtrees for reinsertion of nodes. ■

The recursive method *adjustSubquadrant* takes as its parameters *parent*, the parent of a subtree, *quadrant*, the quadrant of the parent in relation to the deleted node, *subquad*, the index of the parent's subquadrant that is to be adjusted, *winningQuadrant*, the quadrant of the replacement node in relation to the deleted node, and *destination*, the root of the subtree into which any nodes from the subtree should be reinserted. If the subtree is not empty, then there are two cases: (1) if the root of the subtree is in the danger zone, then the entire subtree is reinserted, or (2) the root is not in the danger zone and the method is applied recursively to the two subquadrants in the direction of the danger zone.

```
void adjustSubquadrant(QTNode parent, int quadrant, int subquad,
    int winningQuadrant, QTNode destination) {
    QTNode subtreeRoot = parent.child(subquad);
    if (subtreeRoot != null && !subtreeRoot.isFrontier()) {
        if (dangerZone.inRange(subtreeRoot))
            reinsertEntireSubtree(subtreeRoot, destination);
        else {
            for (int q = 0; q < 4; q++)
                if (q != quadrant && q != winningQuadrant)
                    adjustSubquadrant(subtreeRoot, quadrant, q,
                        winningQuadrant, destination);
        }
    }
}
```

Correctness Highlights: The method performs a preorder traversal beginning at *subtreeRoot*. Clearly any node examined that is in the danger zone is reinserted. Furthermore, for each node encountered that is not in the danger zone, all children that could potentially be in the danger zone (i.e., quadrants in the direction of the deleted node) are recursively visited. ■

48.6.5 Locator Initializers

The method *traverseForVisitor* takes *v*, a visitor, and traverses the quad tree on behalf of the visitor. The work of the method is delegated to root quad tree node, which provides a recursive method for this purpose.

```
protected void traverseForVisitor(Visitor<? super E> v) throws Exception {
    root.traverseForVisitor(v);
}
```

A custom iterator is not provided by the quad tree implementation. Instead, the *iterator* method returns a visiting iterator (Section 8.5) that exploits the recursive algorithm provided in *traverseForVisitor* as the basis for iteration over the elements. Users should note the caveat on page 102.

```
public Locator<E> iterator() {
    return new VisitingIterator();
}
```

Because the quad tree does not provide a custom locator, the *getLocator* method throws an *UnsupportedOperationException*. The *iterator* method should be used to iterate through the collection.

```
public Locator<E> getLocator(E element) {
    throw new UnsupportedOperationException();
}
```

48.7 Performance Analysis

The expected height for a quad tree is $O(\log n)$ under the assumption that the elements are placed into the quad tree in a random order. The minimum possible height is $\lceil \log_4 n \rceil$, and in the worst case the height could be linear. When elements are inserted in a random order, a k -dimensional extension of a quad tree can be viewed as a k -d tree in which 2^k consecutive levels are compressed into one. The cost of searching in a quad tree is bounded by the height of the tree, so it has expected logarithmic cost. A range query takes $O(r + \sqrt{n})$ time, where r is the number of elements in the collection returned. In contrast, for a binary search tree the cost is $O(r + \log n)$. Intuitively, each search in a quad tree can be thought of as being a search in a binary search tree of height $\lceil \log_4 n \rceil$. Thus, the number of leaf nodes that must be searched is $2^{\log_4 n} = \sqrt{n}$.

48.8 Quick Method Reference

QuadTree Public Methods

- p. 767 *QuadTree*()
- p. 767 *QuadTree*(XYComparator<? super E> comp)
- p. 98 void **accept**(Visitor<? super E> v)
- p. 479 void **add**(E element)
- p. 100 void **addAll**(Collection<? extends E> c)
- p. 100 void **clear**()
- p. 97 boolean **contains**(E value)
- p. 99 void **ensureCapacity**(int capacity)
- p. 476 E **get**(int r)
- p. 96 int **getCapacity**()
- p. 96 Comparator<? super E> **getComparator**()
- p. 98 E **getEquivalentElement** (E target)
- p. 782 Locator<E> **getLocator**(E element)
- p. 96 int **getSize**()
- p. 96 boolean **isEmpty**()
- p. 781 Locator<E> **iterator**()
- p. 478 E **max**()
- p. 768 E **max**(int dimension)
- p. 477 E **min**()
- p. 769 E **min**(int dimension)
- p. 479 boolean **remove**(E element)
- p. 100 void **retainAll**(Collection<E> c)
- p. 97 Object[] **toArray**()

- p. 97 E[] **toArray**(E[] array)
- p. 98 String **toString**()
- p. 99 void **trimToSize**()
- p. 771 Collection<E> **withinBounds**(E cornerA, E cornerB)
- p. 771 void **withinBounds**(E cornerA, E cornerB, Collection<? super E> elements)

QuadTree Internal Methods

- p. 781 void **adjustSubquadrant**(QTreeNode parent, int quadrant, int subquad, int winningQuadrant, QTreeNode destination)
- p. 780 void **adjustSubtree**(QTreeNode deletedNode, QTreeNode winner, int winningQuadrant)
- p. 778 int **chooseReplacementNode**(QTreeNode node)
- p. 97 int **compare**(E e1, E e2)
- p. 768 int **conjugate**(int q)
- p. 768 QTreeNode **createTreeNode**(E data)
- p. 97 boolean **equivalent**(E e1, E e2)
- p. 777 void **filterCandidates**()
- p. 771 TreeNode **find**(E target)
- p. 772 QTreeNode **findInsertPosition**(QTreeNode subtree, E element)
- p. 476 int **getLastNodeSearchIndex**()
- p. 777 void **identifyReplacementCandidates**(QTreeNode node)
- p. 772 QTreeNode **insert**(E element)
- p. 772 QTreeNode **insertNode**(QTreeNode node, QTreeNode subtree)
- p. 768 int **leftNeighbor**(int q)
- p. 477 TreeNode **leftmost**()
- p. 476 TreeNode **leftmost**(TreeNode x)
- p. 768 E **max**(E candidate, QTreeNode subtree, int dimension)
- p. 769 E **min**(E candidate, QTreeNode subtree, int dimension)
- p. 779 void **reinsertEntireSubtree**(QTreeNode subtreeRoot, QTreeNode destinationRoot)
- p. 779 void **remove**(TreeNode toDelete)
- p. 768 int **rightNeighbor**(int q)
- p. 477 TreeNode **rightmost**()
- p. 477 TreeNode **rightmost**(TreeNode x)
- p. 99 void **traverseForVisitor**(Visitor<? super E> v)
- p. 478 void **traverseForVisitor**(Visitor<? super E> v, TreeNode x)
- p. 770 void **withinBounds**(QTreeNode node, Box bounds, Box space, Collection<? super E> elements)
- p. 98 void **writeElements**(StringBuilder s)
- p. 479 void **writeElements**(StringBuilder sb)

QuadTree.Box Internal Methods

- p. 765 Box()
- p. 766 Box(E a, E b)
- p. 765 Box(QTreeNode a, QTreeNode b)
- p. 765 Box(double minx, double miny, double maxx, double maxy)
- p. 766 boolean **contains**(E e)
- p. 766 boolean **contains**(double x, double y)
- p. 767 boolean **inRange**(QTreeNode node)
- p. 767 boolean **inRange**(double x, double y)
- p. 766 boolean **intersects**(Box b)
- p. 766 Box **set**(double minx, double miny, double maxx, double maxy)
- p. 766 void **setCorners**(E a, E b)

QuadTree.QTreeNode Public Methods

- p. 764 void **replaceBy**(QTreeNode replacement)
- p. 764 void **resetParent**()
- p. 765 void **traverseForVisitor**(Visitor<? super E> v)

QuadTree.QTNode Internal Methods

- p. 762** *QTNode*(E data)
- p. 763** int **capacity**()
- p. 763** *QTNode* **child**(int index)
- p. 763** E **data**(int index)
- p. 765** *QTNode* **deepestInSubtree**(int quadrant)
- p. 763** int **indexOf**(*QTNode* child)
- p. 763** boolean **isFrontier**()
- p. 764** void **setChild**(int index, *QTNode* child)
- p. 763** int **size**()

Chapter 49

Tagged Collection ADTs

Algorithmically positioned collections, such as ordered collections, priority queues, and spatial collections, use information about the elements to determine their proper placement within the data structure. Often, this information is extracted from the elements themselves with the help of a comparator, a digitizer, or other similar mechanism. However, in many cases it is desirable to organize elements *not* on the basis of information derived from the elements, but instead on the basis of information that is externally assigned to each element by the application. In other words, the application provides, for each element, an external **tag** that is to be associated with that element in the collection. A **tagged collection** is a collection that provides the necessary support for associating these tags with the corresponding elements. Most tagged collections permit duplicate tags. For example, in a collection of jobs, the application may assign to each element a tag denoting the priority of that job. There may be many jobs with the same priority. However, some tagged collections enforce the restriction that tags are unique. Each tag in such a collection is called a **key**, because it can be presented to the collection in order to find its associated element. However, even when the tags are unique, multiple keys may map to the same element.

Each insertion into a tagged ADT creates an association between the tag and data object. Since the tag determines the placement of the element within the collection, the tag should be immutable, for modifying it could violate the ordering properties within the data structure. Any replacement of the tag associated with an element should be carried out by calling a method on the tagged collection. In some cases, the tag may be the value of a field of the associated element, but typically the tag is not part of the element.

The tagged collection ADTs and their implementations closely parallel those for the untagged collections. In fact, the only significant difference is that the tag, instead of the element itself, is used to arrange the elements. Therefore, rather than redefine these data structures from scratch, we introduce a **tagged element** class that simply pairs a tag and an associated element. The element held in a tagged element can be *null*, or any other object, such as a boxed primitive data value, an entire data structure, or a location in secondary storage. Having defined the tagged element type, we can use the implementation of an existing untagged collection as the underlying implementation of a tagged collection, by using tagged elements as wrapped collection's element type. However, this approach demands that care be taken in the treatment of object equivalence and comparison. In the interface of the tagged collection ADTs, it will be most natural for application developers to define their comparators only in terms of the tag type, and not in terms of the tagged elements as a whole. However, since the underlying data structure will store tagged elements and is designed to accept comparators that do compare its elements as a whole, our design must provide a way to “trick” the underlying data structure into using the application-provided comparator to compare tagged elements using only the tags. For this purpose, we will define a general-purpose tagged element comparator that wraps the application-provided comparator, and uses that wrapped comparator to compare tagged elements only in terms of their tags.

In our design, the tagged collections wrap the untagged collections, but an alternative design would be to implement the tagged collections first, and then let the untagged collections simply ignore the “element” portion of each tagged element. This is the approach taken in the `java.util`

collections, where the tagged element is a `Map.Entry` containing a key and a value. The associated untagged data structures use the “key” portion of each map entry to hold the element, and the “value” part is ignored. A disadvantage of that approach is that the unnecessary map entry objects and empty value slots waste space for untagged collections.

49.1 Tagged Element

package `collection.tagged`

TaggedElement`<T,E>`

Each element and its associated tag is stored in a tagged collection as a **tagged element**. A tagged element also includes methods to compare and retrieve the tags and data elements.

Instance Variables and Constants: Each *TaggedElement* instance has two instance variables, *key*, which is the key, and *element*, which is the associated value for that key.

```
protected T tag;
protected E element;
```

The constructor takes *t*, the tag, and *e*, the element, and initializes the tagged element to hold these values. The method requires that the tag is not *null*.

```
public TaggedElement(T t, E e) {
    if (t == null)
        throw new IllegalArgumentException(“Tag cannot be null”);
    tag = t;
    element = e;
}
```

A protected no-argument constructor is provided for subclasses that do not wish to provide initial values for the tagged element.

```
protected TaggedElement() {}
```

Each tagged collection data structure wraps the corresponding collection data structure to hold tagged elements. However, the semantics for the tagged collection dictates that the tag, as opposed to the entire entry, is used for organizing the data. To support this semantics, the *hashCode* method returns a hashcode based on only the tag.

```
public int hashCode() {return tag.hashCode();}
```

While the organization of the data in tagged collection is based only on the tag, the application program needs methods to access both the tag and associated element. The *getTag* method returns the tag from this tagged element.

```
public T getTag() {return tag;}
```

The *getElement* method returns the element from this tagged element.

```
public E getElement() {return element;}
```

The *setElement* method takes *e*, the new value for the element field.

```
public void setElement(E e) {element = e;}
```

The *toString* method returns a string of the form, *tag* $\rightarrow$ *element*.

```
public String toString(){return tag + “  $\rightarrow$  ” + element;}
```

The remainder of this section presents several classes that support the use of tagged elements within various types of tagged collections. One additional such class, the *PQTaggedElement*, is defined as an inner class of the tagged priority queue implementation (Section 49.8.1).

49.1.1 Mutable Tagged Element

Recall that the underlying implementations will be ordinary (i.e., untagged) collections, each of which will happen to hold tagged elements. Since these underlying collections know only about their element types, all of their methods are defined in terms of that type. In particular, the search methods in these underlying collections will expect a target element whose type matches the element type of the collection: they will *not* expect a tag, but instead will expect a tagged element. Since the user of the tagged collection will search on the basis of tags, we need a mechanism to put that tag into a target tagged element that can be passed to the underlying collection. To enable reuse of such tagged elements, it is necessary to allow the tag field, which is normally immutable, to be reassigned for each search. To support this need without exposing the tag of all tagged elements for mutation, we extend the *TaggedElement* class to define the following *MutableTaggedElement* class.

```
public class MutableTaggedElement<T, E> extends TaggedElement<T, E> {
    public TaggedElement<T, E> setTag(T t) {
        tag = t;
        return this;
    }
}
```

49.1.2 Tagged Element Comparator

We compare tagged elements by wrapping a provided comparator defined over only the tags. That is, our comparator wraps a comparator of type *T* and supports comparison of tagged elements whose tag type is *T*.

```
public class TaggedElementComparator<T> implements Comparator<TaggedElement<T, ?>> {
    Comparator<? super T> comp;

    public TaggedElementComparator(Comparator<? super T> comp) {
        this.comp = comp;
    }

    public int compare(TaggedElement<T, ?> a, TaggedElement<T, ?> b) {
        return comp.compare(a.tag, b.tag);
    }
}
```

```

    public Comparator<? super T> getTagComparator() {
        return comp;
    }
}

```

49.1.3 Tagged Element Digitizer

The tagged digitized ordered collection data structures require digitizers for tagged elements that depend only on the tags. The following tagged element digitizer, defined in the package *collection.tagged.ordered.digitized*, wraps a digitizer defined for a tag type to create a digitizer defined for a corresponding tagged element type.

```

public class TaggedElementDigitizer<T> implements Digitizer<TaggedElement<T, ?>> {
    Digitizer<? super T> digitizer;

    public TaggedElementDigitizer(Digitizer<? super T> digitizer) {
        this.digitizer = digitizer;
    }

    public int getBase() {
        return digitizer.getBase();
    }

    public int getDigit(TaggedElement<T, ?> x, int place) {
        return digitizer.getDigit(x.getTag(), place);
    }

    public boolean isPrefixFree() {
        return digitizer.isPrefixFree();
    }

    public int numDigits(TaggedElement<T, ?> x) {
        return digitizer.numDigits(x.getTag());
    }
}

```

49.1.4 Tagged Element XY Comparator

The quad tree implementation requires an XYComparator over its elements. To support tagged quad trees, we define the following comparator for tagged elements, where the tag is two-dimensional point. It wraps the XYComparator that is defined over tags that are two-dimensional points.

```

public class TaggedElementXYComparator<T extends XYPoint>
    implements XYComparator<TaggedElement<T, ?>> {

    XYComparator<? super T> comp;

    public TaggedElementXYComparator(XYComparator<? super T> comp) {
        this.comp = comp;
    }
}

```

```

public double compareX(TaggedElement<T, ?> a, TaggedElement<T, ?> b) {
    return comp.compareX(a.getTag(), b.getTag());
}

public double compareY(TaggedElement<T, ?> a, TaggedElement<T, ?> b) {
    return comp.compareY(a.getTag(), b.getTag());
}

public double getX(TaggedElement<T, ?> item) {
    return comp.getX(item.getTag());
}

public double getY(TaggedElement<T, ?> item) {
    return comp.getY(item.getTag());
}

//uses x-coordinate by default
public int compare(TaggedElement<T, ?> a, TaggedElement<T, ?> b) {
    return comp.compare(a.getTag(), b.getTag());
}

public int quadrant(TaggedElement<T, ?> origin, TaggedElement<T, ?> item) {
    return comp.quadrant(origin.getTag(), item.getTag());
}
}

```

49.2 Tagged Collection Interface

package collection.tagged

Iterable<*TaggedElement*<*T*,*E*>>

└ **TaggedCollection**<*T*,*E*>

The *TaggedCollection* interface parallels the *Collection* interface. To reflect that elements are inserted with an application-specified tag, the *add* method from the *Collection* interface is replaced by a *put* method that takes a tag and an element. Likewise, the *Collection* *addAll* method is replaced *putAll*. Excluding the fundamental distinction that the tag is used to organize the elements, all other methods have the same semantics as those of the *Collection* interface.

TaggedCollection(): Creates a new empty tagged collection.

void accept(Visitor<? super TaggedElement<T,E>> v): Traverses each tagged element of this collection, in the iteration order, on behalf of the visitor.

void clear(): Removes all tagged elements from this collection.

boolean contains(T tag): Returns *true* if a tagged element with an equivalent tag exists in this collection. Otherwise *false* is returned.

Iterator<E> elements(): Returns an iterator over the elements that has been initialized to just before the first element in the iteration order. The iteration order of the elements is defined based on the iteration order of the associated tags.

void ensureCapacity(int capacity): Increases the capacity of this tagged collection, if necessary, to ensure that it can hold at least *capacity* tagged elements.

E get(T tag): Returns an element with the given tag. This method requires that *tag* be in use. It throws a *NoSuchElementException* otherwise.*

int getCapacity(): Returns the current capacity of this tagged collection.

Locator<TaggedElement<T,E>> getLocator(E target): Returns a locator that has been initialized to a tagged element with an equivalent tag. Like the *iterator* method, this method enables navigation, but from a specified starting point. This method throws a *NoSuchElementException* if there is no tagged element with an equivalent tag in the collection.

int getSize(): Returns the number of tagged elements, *n*, in this collection.

boolean isEmpty(): Returns *true* if this collection contains no elements, and otherwise returns *false*.

Iterator<TaggedElement<T,E>> iterator(): Returns an iterator that has been initialized to FORE. The returned iterator can be used to navigate within the collection and to remove the element currently associated with the iterator. In general, no assumption is made about the **iteration order**, the order in which the iterator advances through the collection. However, some tagged collections specify a particular iteration order based on the values of the tags.

void put(T tag, E element): This method creates a new tagged element with the given values and inserts this tagged element into this collection. An *AtCapacityException*, an unchecked exception, is thrown if the collection is already at capacity.

void putAll(TaggedCollection<T,E> tc): Adds all tagged elements in *tc* to this tagged collection.

E remove(T tag): Removes, from this tagged collection, a tagged element with the given tag. Returns the element that was removed, and throws a *NoSuchElementException* when there is no element with an equivalent tag.

Iterator<T> tags(): Returns an iterator over the tags that has been initialized to just before the first tag in the iteration order.

String toString(): Returns a string that describes each tagged element in the collection, in the iteration order.

void trimToSize(): Trims the capacity of this tagged collection to be its current size. An application can use this operation to minimize the space usage when future insertions are unlikely.

*Note that this semantics differs from the Java API, where *get* returns *null* when there is no element in the collection with the given tag. This approach can create confusion between a tag that maps to *null* and the tag being absent. The Java documentation indicates that *containsKey* (semantically equivalent to our *contains* method) should be used to distinguish those two cases. Our specification requires that the parameter to *get* be a tag in use, and that *get* throws an exception when it is not. This eliminates the potential for program errors that could result from a return value with two possible interpretations.

Collection<E> values(): Returns a collection of all the elements held in this tagged collection. Adding (or removing) elements to (or from) the returned collection does not change the tagged collection.

49.3 Tracked Tagged Interface

package collection.tagged

TaggedCollection<T,E>
└ *TrackedTagged*<T,E>

Similar to the *Tagged* interface, the *TrackedTagged* interface includes a method to insert a tagged element into a tagged collection that returns a tracker for it.

Locator <TaggedElement<T,E>> putTracked(T tag, E element): This method creates a new tagged element for the given tag and element, and inserts it into this collection. A locator that tracks the new tagged element is returned. An *AtCapacityException*, an unchecked exception, is thrown if the collection is already at capacity.

49.4 Competing ADTs

We briefly discuss other ADTs that may be appropriate in a situation when a tagged collection is being considered.

Collection: If the property used to organize the elements is directly defined over the elements themselves, then duplicating this portion of the element as a tag increases the space usage, and an untagged collection (Chapter 7) should be considered. However, there are some situations where duplication of the tag is warranted. For example, if extracting the information from the element is expensive, it makes sense to create the tag once, rather than use an untagged collection and force the comparator to extract that information every time a comparison is made. Duplicating the tag is also warranted when the elements are very large and the tags are relatively small. If the collection itself is large, then each element will most likely be kept in secondary storage. In such a situation, associating a tag with a location on secondary storage enables a more efficient search.

Tagged Bucket Collection: If the decision has been made to associate a tag with each element, another question is whether or not the elements that share a tag should be grouped together. The advantage of grouping elements with a shared tag is that the search cost depends on the number of unique tags, versus the number of tagged elements. Furthermore, grouping the elements enables efficient access to all elements with a given tag. When it is expected that more than just a small fraction of elements will share a tag, a tagged bucket collection (Chapter 50) is a better choice.

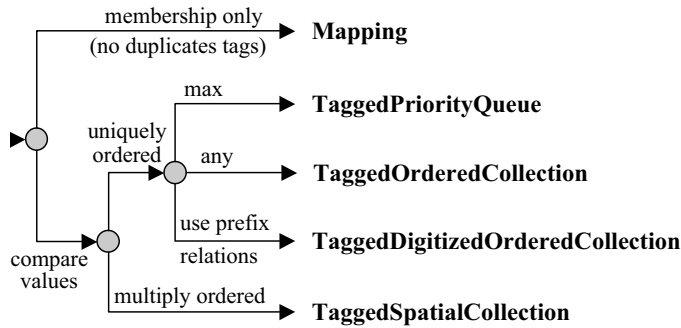


Figure 49.1

A decision tree for selecting a tagged collection ADT.

49.5 Selecting a Tagged Collection ADT

Having determined that a tagged collection is more appropriate than a collection or a tagged bucket collection, one must decide which of the following TaggedCollection ADTs best supports the application. This decision depends upon the ways in which the tag will be used to access the data, and the decision process mirrors that for choosing an algorithmically positioned Collection ADT, described in detail in Section 2.6.2. Figure 49.1 shows the portion of the decision tree of Figure 2.6 that pertains to selecting among the tagged collection ADTs discussed in this book.

The following ADTs implement the tagged collection interface. Each tagged collection is a variation of the corresponding untagged ADT (where a Mapping is a shorthand for a **TaggedSet**).

Mapping - This ADT is the tagged version of the Set ADT (Chapter 20). Since each tag in a mapping is unique, the tag is called a **key**. The key is used only to identify the element. This ADT is sometimes called a **dictionary** since it is used to lookup an element based on its key. However, unlike an alphabetized dictionary for looking up the definitions of words, keys in a mapping are not ordered. The primary methods are to put in a new tagged element, remove a tag (and its associated element), and to get the element associated with a given tag. Mapping data structures typically support these methods in expected constant time.

TaggedPriorityQueue - A tagged priority queue is intended for applications in which efficient access is required for the element with the highest “priority” tag. The ordering among the tags determines the priority among the elements’ tags where $t_1 > t_2$ implies that the element with tag t_1 has a higher priority than the element associated with tag t_2 . While there is an order defined over the tags, iteration need not follow this order. Also, data structures for a tagged priority queue typically require linear time to search for a specific an element with a specified tag. If the application requires efficient access to the element with the lowest priority (instead of the highest), a reverse comparator (Section 5.2.3) can be used to wrap the usual comparator.

TaggedOrderedCollection - This ADT is designed for applications in which iteration must correspond to the ordering of the tags, as defined by the comparator. The tags need not be unique. While not as efficient as a mapping, most tagged ordered collection implementations can add a tagged element, remove a tagged element, and find an element based on its tag in logarithmic time. In addition, there are efficient methods to find the element with the minimum tag (first in the iteration order), the element with the maximum tag (last in the iteration order), and to move forward or backwards in the iteration order. Thus, an application program can

efficiently implement a **range search** (find all elements with tags in a certain range), or find the element in the iteration order that has a tag closest to a desired value.

TaggedDigitizedOrderedCollection - This ADT orders elements based on an application-specific digitizer (Section 5.3) defined over the tags. This ADT supports all the methods of the TaggedOrderedCollection ADT. In addition, it provides methods to (1) find all tagged elements, whose tags share the longest common prefix with a given tag, over all elements in the collection, and (2) to find all tagged elements whose tag has a given prefix. Most of the data structures we present for maintaining tagged digitized ordered collections guarantee a worst-case search time proportional to the length of the prefix that distinguishes the given tag from all other tags used in the collection.

TaggedSpatialCollection - This ADT supports multiply ordered tags, and should be used when the application must efficiently find the set of tagged elements with tags in a certain range in each dimension. For example, a set of mobile devices could be tagged with a location represented as longitude and latitude coordinates. One ordering for the tags is based on the longitude and a second ordering is based on the latitude. The application may need to locate all devices in a rectangular geographic region. In general, multiply ordered tags need not be truly spatial as in this example, but many problems still reduce to finding all tagged elements with a tag in a specified d -dimensional box.

49.6 Tagged Collection Wrapper

package collection.tagged

TaggedCollectionWrapper<T,E> implements *TaggedCollection*<T,E>

Internal Representation: Rather than implement each of the tagged collections from scratch, we define a tagged collection wrapper that can be used to transform any untagged collection into a tagged collection. Each element in the collection is a tagged element. For a tagged collection to have the specified behavior, it is required that the equivalence tester over a tagged element be defined according to the tag.

Critical Mutators: *clear, ensureCapacity, put, putAll, remove, trimToSize*

Instance Variables and Constants: The tagged collection wrapper has two instance variables: *pairs* which is the wrapped collection of tagged elements, and *target*, which holds the target tagged element for searches.

```
protected Collection<TaggedElement<T, E>> pairs;
protected MutableTaggedElement<T, E> target = new MutableTaggedElement<T, E>();
```

When the application needs to search based on a tag, the underlying (untagged) collection expects an entire tagged element to be provided as a search target. It is for this purpose that we create a mutable tagged element and save it in the instance variable *target*. This avoids the need to create a new tagged element for the target of each search. In a multithreaded application, any method that depends upon *target* must be synchronized to ensure that no method call by another thread changes *target* during a search. We show the methods this way for illustration.

Abstraction Function: Let TC be a tagged collection instance. The abstraction function

$$AF(TC) = \{(t, v) \mid \exists te \in \text{pairs and } te.tag = t \text{ and } te.element = v\}.$$

Constructors

The constructor for the tagged collection wrapper takes *pairs*, the collection to wrap. It throws an *IllegalArgumentException* when the provided collection is not empty.

```
public TaggedCollectionWrapper(Collection<TaggedElement<T, E>> pairs) {
    if (!pairs.isEmpty())
        throw new IllegalArgumentException("provided collection must be initially empty");
    this.pairs = pairs;
}
```

TrivialAccessors

All of the trivial accessors are delegated to the wrapped collection implementation.

```
public int getCapacity() {return pairs.getCapacity();}
public int getSize() {return pairs.getSize();}
public boolean isEmpty() {return pairs.isEmpty();}
```

Algorithmic Accessors

The *contains* method takes *tag*, the target tag. It returns *true* if and only if there is some tagged element in this collection with an equivalent tag to the target.

```
public synchronized boolean contains(T tag) {
    return pairs.contains(target.setTag(tag));
}
```

The *get* method takes *tag*, the target tag, and returns a tagged element with an equivalent tag. It throws a *NoSuchElementException* when there is no element with an equivalent tag.

```
public synchronized E get(T tag) {
    return pairs.getEquivalentElement(target.setTag(tag)).getElement();
}
```

The *getLocator* method takes *tag*, the target tag. It returns a locator that has been placed at a tagged element with an equivalent tag. It throws a *NoSuchElementException* when there is no element with an equivalent tag.

```
public synchronized Locator<TaggedElement<T, E>> getLocator(T tag) {
    return pairs.getLocator(pairs.getEquivalentElement(target.setTag(tag)));
}
```

The *values* method returns a collection of all elements in this tagged collection. This implementation creates a positional collection (specifically, an array), and the elements are inserted in the iteration order.

```

public Collection<E> values() {
    Collection<E> coll = new Array<E>(getSize());
    for (TaggedElement<T, E> te : this)
        coll.add(te.element);
    return coll;
}

```

The *toString* method returns a string that describes each tagged element in the collection.

```

public String toString() {
    return pairs.toString();
}

```

The *accept* method takes *v*, a visitor. It traverses the entire collection on behalf of a visitor. It throws a *VisitAbortedException* when the traversal is aborted due to an exception raised by the visitor, in which case the cause held by the *VisitAbortedException* is the exception thrown by the visitor.

```

public void accept(Visitor<? super TaggedElement<T, E>> v) {
    pairs.accept(v);
}

```

Representation Mutators

Recall that *ensureCapacity* takes *capacity*, the desired capacity for the collection, and increases the capacity of the collection, if necessary, to ensure that it can hold at least *capacity* elements.

```

public void ensureCapacity(int capacity) {
    pairs.ensureCapacity(capacity);
}

```

The *trimToSize* method trims the capacity of this collection to be its current size. An application can use this operation to minimize space usage.

```

public void trimToSize() {pairs.trimToSize();}

```

Content Mutators

The *put* method takes *tag*, the tag for the tagged element to add, and *data*, the associated data. It creates a new tagged element with the given values and inserts this tagged element into this collection. It throws an *AtCapacityException* when the collection is already at capacity.

```

public synchronized void put(T tag, E data) {
    pairs.add(new TaggedElement<T, E>(tag, data));
}

```

The *putAll* method takes *tc*, the tagged collection to be added to this collection.

```

public synchronized void putAll(TaggedCollection<T, E> tc) {
    pairs.addAll((Collection<? extends TaggedElement<T, E>>) tc);
}

```

The *remove* method takes *tag*, the target tag. It removes some tagged element in this collection with an tag equivalent to the given tag, and returns the removed element. It throws a *NoSuchElementException* when there is no element with an equivalent tag.

```

public synchronized E remove(T tag) {
    Locator<TaggedElement<T, E>> loc = pairs.getLocator(target.setTag(tag));
    Object x = loc.get();
    E elem = loc.get().getElement();
    loc.remove();
    return elem;
}

```

Correctness Highlights: By the correctness of *setTag* and the *getLocator* method of the wrapped collection implementation, the exception is appropriately thrown when there is no element with the equivalent tag. The rest of the correctness follows from the correctness of the locator *get* and *remove* methods, and the tagged element *getElement* method. ■

A disadvantage of this implementation is that each call to *remove* creates a locator to find and remove the element. As an optimization, one could provide a locator method that repositions a locator at a given element so that the locator could be reused. Alternatively, one could provide a *remove* method in *Collection* that removes an element and returns the equivalent element that was removed, throwing a *NoSuchElementException* as required. In that case, the body of the following method could be replaced by the single line

```

return pairs.remove(target.setTag(tag));

```

The *clear* method removes all elements from this collection.

```

public void clear() {
    pairs.clear();
}

```

Locator Initializers

The *iterator* method creates a new iterator at FORE.

```

public Iterator<TaggedElement<T, E>> iterator() {
    return pairs.iterator();
}

public Iterator<T> tags() {
    return new Iterator<T>() {
        Iterator<TaggedElement<T, E>> wrapped = iterator();
        public boolean hasNext() { return wrapped.hasNext(); }
        public T next() { return wrapped.next().tag; }
        public void remove() { wrapped.remove(); }
    };
}

public Iterator<E> elements() {
    return new Iterator<E>() {
        Iterator<TaggedElement<T, E>> wrapped = iterator();
        public boolean hasNext() { return wrapped.hasNext(); }
    };
}

```

```

    public E next() { return wrapped.next().element; }
    public void remove() { wrapped.remove(); }
};
}

```

49.7 Mapping ADT

package collection.tagged.mapping

Iterable<*TaggedElement*<*K*,*E*>>

└ *TaggedCollection*<*K*,*E*>

└ *Mapping*<*K*,*E*>

The Mapping ADT, is a tagged variation of the Set ADT. While the equivalence tester provided for a set might use a single field within the element (that can be viewed as a tag), in a mapping there is an explicit association created from the tag to the associated data element. Also, the equivalence of two elements is defined based on the equivalence between the tags. Likewise the *hashCode* method for a tagged element is based solely on the tag. Just as a Set requires that its elements be unique, a Mapping requires that its tags be unique, so we refer to its tags as **keys**.

Example Application for a Mapping

One very common application of the Mapping ADT is to associate objects for efficient lookup, particularly when it would be infeasible for the key to have an instance variable that refers to the associated element. A standard application for the Mapping ADT is for databases and information systems. For example, given a student id one may want to look up additional information about the student, or given the bar code of an item in a store, one may want to look up product and inventory information.

Mappings are also useful as an efficient bridge between two data structures. For example, one might have two data structures storing a collection of elements in different ways, and one may want to remove an element from the second collection whenever its corresponding element is removed from the first. A mapping from elements to trackers could be used to associate each element in the first collection with a tracker in the second, enabling constant time access for removal.

Another example arises in our implementation for the Graph ADT and Weighted Graph ADT, where we use a mapping to maintain a variety of information associated with a vertex in a graph.

Interface

While no new methods are added to the *TaggedCollection* interface for a mapping, the semantics of *get*, *put*, *putAll*, and *putTracked* are slightly different. The distinction is that any data structure that implements the *Mapping* interface is expected to guarantee that no two tags held in the mapping are equivalent. Since the tags are unique in a mapping, we refer to each tag as a **key**. Each key serves as an identifier for its associated element. A key object should not be modified while it is in use in a mapping, for any such modification could violate invariants about the placement of tagged elements within the data structure.

Like the Set ADT, if the call *put*(*k*,*v*) is made when there is already a mapping entry (*k'*,*v'*) in the mapping with *k* $\cong$ *k'*, then tagged element (*k'*,*v'*) is replaced by (*k*,*v*). So *get* and *put* have the following semantics.

E get(T key): Returns the most recent value inserted in the tagged collection with the given key.

void put(T key, E element): This method creates a new tagged element with the given key and element. If a tagged element already exists with the given key, then the tagged element is replaced. An *AtCapacityException* (an unchecked exception) is thrown when the collection is at capacity and the key was not already in use.

The *putAll* and *putTracked* methods have analogous semantics.

Optimizations: In typical uses of a mapping, it is frequently the case that one calls *contains* on a tag to determine if it is in the map, and then immediately calls *get* to retrieve the associated data. By caching the location in the data structure for the most recent find (caused by *contains*), one could avoid the cost of the second search (for *get*) in many cases.

Competing ADTs

We briefly discuss ADTs that may be appropriate in a situation when a Mapping ADT is being considered.

PositionalCollection ADT: If the key is an application-specified position (i.e., index) of an element in a line and that position must be updated as new elements are inserted and deleted, then a PositionalCollection ADT is a better choice.

Set ADT: If the object used for the tag is contained within the associated data element, and it is appropriate to define an equivalence tester based on it, then a Set ADT should be considered since it slightly reduces the space usage. However, it is important to consider whether a Set ADT or Mapping ADT is more natural for the application.

BucketMapping ADT: If each element is identified by tag, yet there could be multiple elements with the same tag, then a bucket mapping is a better option. Along with allowing multiple elements with the same tag, an advantage of a bucket mapping over a mapping is that the complexity of the primary methods depends on the number of unique tags, as opposed to the number of tagged elements in the collection.

Hierarchy of Mappings: If the number of unique tags stored in the mapping is so large that the data structure cannot fit in RAM, then either the virtual memory can be used or a hierarchy of mappings could be introduced. For example if the key is a phone number, then there could be a mapping where the key is the area code, and the associated data is a mapping from the remainder of the phone number to the associated data for the entire phone number. Additional levels of the hierarchy can be added until each mapping is small enough to fit into memory.

TaggedDigitizedOrderedCollection ADT: While a tagged digitized ordered collection is most commonly used when there is an ordering defined among the keys, it can also be used to implement a mapping. If the keys can be viewed as a sequence of digits, and keys are often distinguishable after looking at relatively few digits (as is often the case with an unsuccessful search), then the TaggedDigitizedOrderedCollection ADT is more appropriate than a mapping. For tags that are naturally defined as a sequence of digits (e.g., words), using a tagged digitized ordered collection avoids the need to make potentially expensive comparisons among the tags since a digitizer makes use of a canonical ordering of the digits, and needs to examine only the digits of the tag necessary to distinguish it from the other tags in use.

Selecting a Mapping Data Structures

The tagged collection wrapper implements all of the *Mapping* methods in terms of a wrapped set instance. The process of selecting which Set data structure to wrap parallels the process described in Section 20.6 on page 283. Table 49.2 provides a high-level summary of the trade-offs among the Mapping ADT data structures.

Direct Addressing Mapping: This data structure provides excellent performance but $O(|\mathcal{U}|)$ space is required, where $\mathcal{U}$ is the universe of possible keys. When n is expected to be much smaller than $|\mathcal{U}|$ then either a separate chaining mapping or an open addressing mapping should be used.

Separate Chaining Mapping: This data structure is applicable when only a small fraction of the elements in $\mathcal{U}$ will be stored in the collection. It is the right choice when deletions occur frequently, or if n changes dramatically since it can tolerate large swings in the size without the need to resize.

Open Addressing Mapping: This data structure is also applicable when only a small fraction of the elements in $\mathcal{U}$ will be stored in the collection. It can make very good use of a small amount of space, but needs to be resized more frequently than does separate chaining. Also, the time complexity of the primary methods grows as the number of recent deletions grow. Thus, if a lot of deletions are being made, then a separate chaining mapping (or, if applicable, a direct addressing mapping) should be used.

49.7.1 Direct Addressing Mapping

The tagged collection wrapper provides all methods except for the constructor. The constructor for this class takes *capacity*, the desired capacity. It creates a hash table with the specified capacity (plus one to handle *null* values).

```
public class DirectAddressingMapping<K, V> extends TaggedCollectionWrapper<K, V>
    implements Mapping<K, V> {
    public DirectAddressingMapping(int capacity) {
        super(new DirectAddressing<TaggedElement<K, V>>(capacity));
    }
}
```

49.7.2 Open Addressing Mapping

The most general constructors for the remaining mapping data structures take four arguments: *capacity*, the desired capacity, *load*, the target load, *equivalenceTester*, the comparator that defines equivalence of objects, and *hasher*, the user-supplied hash code computation. They create a hash table with the specified capacity (plus one to handle *null* values), and with the given comparator and hasher. They throw an *IllegalArgumentException* when *capacity* < 0 or *load* ≥ 1.0. Any additional number of convenience constructors that replace some parameters by default values could be added. As an illustration, we show one convenience constructor for *OpenAddressingMapping* that is used in the *WeightedGraph* class (Chapter 56).

```
public class OpenAddressingMapping<K, V> extends TaggedCollectionWrapper<K, V>
    implements Mapping<K, V> {
```

Key				
⦿	Excellent			
○	Very Good			
●	Fair			
•	Method does nothing			
Method		DirectAddressingMapping	SeparateChainingMapping	OpenAddressingMapping
Mapping Methods	clear(), per element	⦿	⦿	⦿
	contains(tag)	⦿	○	○
	ensureCapacity(x)	•	●	●
	get(tag), get Locator(tag)	⦿	○	○
	getTaggedElements(), per element	⦿	⦿	⦿
	put(tag,d), putAll(tc), per element in tc	⦿	○	○
	remove(tag)	⦿	○	○
	accept(v), toString(), values(), per element	⦿	⦿	⦿
Other Issues	trimToSize(), per element	•	⦿	⦿
	space usage	⦿	○	⦿
	much of element universe is expected to be used	✓		
	how remove affects future operations	⦿	⦿	○
Marker Methods	frequency of resizing	•	⦿	○
	advance()	○	○	○
	get()	○	○	○
	hasNext()	○	○	○
	inCollection()	○	⦿	○
	next()	○	○	○
	remove(), retreat()	○	○	○
Time Complexity		Other Issues		
⦿	worst-case O(1) time	⦿	excellent	
○	expected-case O(1) time	○	fair	
●	O(n) time	•	not applicable	
•	method does nothing			

Table 49.2 Trade-offs among the data structures that implement the Mapping ADT. For all three we assume the space usage is at most $4n$ which requires that for direct addressing mapping that $|\mathcal{U}| \leq 3n$.

```

public OpenAddressingMapping(int capacity, double load, Comparator<? super K> equiv) {
    super(new OpenAddressing<TaggedElement<K, V>>(capacity, load,
        new TaggedElementComparator<K>(equiv)));
}

public OpenAddressingMapping(int capacity) {
    this(capacity, OpenAddressing.DEFAULT_LOAD,
        Objects.DEFAULT_EQUIVALENCE_TESTER);
}
}

```

49.7.3 Separate Chaining Mapping

The constructor is similar to that for OpenAddressingMapping. Additional constructors with fewer parameters could be defined for convenience.

```

public class SeparateChainingMapping<K, V> extends TaggedCollectionWrapper<K, V>
    implements Mapping<K, V> {
    public SeparateChainingMapping(int capacity, double load,
        Comparator<? super K> equivalenceTester) {
        super(new SeparateChaining<TaggedElement<K, V>>(capacity, load,
            new TaggedElementComparator<K>(equivalenceTester)));
    }
}

```

49.8 Tagged Priority Queue ADT

package collection.tagged.priority

TaggedCollection<*T*,*E*>
 ↑ *TaggedPriorityQueue*<*T*,*E*>

In this section we present the TaggedPriorityQueue ADT, which is a tagged version of the PriorityQueue ADT. While the comparator used within a priority queue might use a single field of the element (that can be viewed as a tag), in a tagged priority queue there is an explicit association created from the tag to the associated data element. Also, the comparator is required to be based upon only the tags. The tags held within a tagged priority queue need not be unique.

The implementation of the tagged priority queue is slightly more involved than the implementation of the other tagged collections since it must provide a mechanism to support the locator *update* method that changes the tag.

Example Application for a Tagged Priority Queue

We use a tagged priority queue within the greedy tree builder (Section 57.1) of the weighted graph class, which is used by both Dijkstra's shortest path algorithm (Section 57.2) and Prim's minimum spanning tree algorithm (Section 57.3). For this application, the tag is a measure of cost, and the associated data is a vertex. As another example application, suppose each job placed in a print queue

is assigned a priority, and the application wants to efficiently find the job with the highest priority. In this case a `TaggedPriorityQueue` is the appropriate ADT because the ordering is not intrinsic to the object themselves and must be externally added as a tag.

Interface

The following methods are added to those of the *TaggedCollection* interface.

***TaggedElement*<T,E> *extractMax*():** Removes and returns a tagged element with a highest priority tag. It throws a *NoSuchElementException* when this collection is empty.

***PriorityQueueLocator*<*TaggedElement*<T,E>> *getLocator*(T tag):** Returns a priority queue locator that has been initialized at a tagged element with the given tag. It throws a *NoSuchElementException* if there is no element in this collection with an equivalent tag.

***Iterator*<*TaggedElement*<T,E>> *iterator*():** Returns an iterator that has been initialized to FORE.

***TaggedElement*<T,E> *max*():** Returns a tagged element with a highest priority tag. It throws a *NoSuchElementException* when this collection is empty.

***PriorityQueueLocator* <*TaggedElement*<T,E>> *putTracked*(T tag, E element):** This method creates a new tagged element using the given tag and element, and inserts it into this collection. A priority queue locator that tracks the new tagged element is returned. An *AtCapacityException*, an unchecked exception, is thrown if the collection is already at capacity.

***void updateTag*(T tag, *PriorityQueueLocator*<*TaggedElement*<T,E>> loc):** This method replaces the tag of the tagged element at the given locator position by *tag*, and makes any required updates to the underlying data structure.

Competing ADTs

We briefly discuss ADTs that may be appropriate in a situation when a `TaggedPriorityQueue` ADT is being considered.

PriorityQueue ADT: If the object used for the tag is contained within the associated data element, and it is appropriate to define an comparator based on it, then a `PriorityQueue` ADT should be considered since it slightly reduces the space usage. However, it is important to consider whether a `PriorityQueue` ADT or `TaggedPriorityQueue` ADT is more natural for the application.

TaggedBucketPriorityQueue ADT: If each element is identified by tag but it is expected that multiple elements will have the same tag, then a tagged bucket priority queue should be considered. The advantage of a tagged bucket priority queue over a tagged priority queue is that the complexity of the primary methods depends on the number of unique tags as opposed to the number of tagged elements in the collection.

TaggedOrderedCollection ADT: If the application must efficiently retrieve an element with a given tag, or requires the iteration order to be based on an ordering defined by the tags, the `TaggedOrderedCollection` ADT is a better choice.

Selecting a TaggedPriorityQueue Data Structures

The tagged priority queue wrapper implements all of the *TaggedPriorityQueue* methods in terms of a wrapped priority queue instance. The process of selecting which priority queue data structure to wrap parallels the process described in Section 24.4 on page 346. Table 49.3 provides a high-level summary of the trade-offs among the data structures we present for the TaggedPriorityQueue ADT.

TaggedBinaryHeap: The tagged binary heap has worst-case logarithmic cost for *put*, *extract-Max* and *update* (through a priority queue locator). Since the lead constant within the asymptotic notation is small, all of the methods run quite quickly. Also the space usage is very good. The main drawback is that merging two tagged binary heaps takes linear time. Also, increasing the priority for the tag of a tracked element takes logarithmic time.

TaggedLeftistHeap: The tagged leftist heap supports *merge* in logarithmic time. In a threaded implementation, all of the methods would have the same time complexity as for a tagged binary heap, except for *merge* which is improved from linear to logarithmic time, and *putAll*, which improves to constant time per new element.

TaggedPairingHeap: The tagged pairing heap is a **self-organizing** data structure in which the amortized cost for *put*, *merge*, and *remove* through a tracker are all logarithmic. Updating the priority for a tracked element has logarithmic amortized cost. The space usage of the tagged pairing heap is similar to that of the tagged leftist heap.

TaggedFibonacciHeap: The tagged Fibonacci heap is a more complex self-organizing data structure. It has amortized constant cost for *put* and *merge*. Increasing the priority of a tagged element has amortized constant cost. Removing a tagged element or lowering the priority of a tagged element has logarithmic amortized cost. A tagged Fibonacci heap yields the asymptotically fastest implementation of Prim's minimum spanning tree algorithm (Section 57.3) and Dijkstra's shortest path algorithm (Section 57.2).

49.8.1 Tagged Priority Queue Wrapper

package collection.tagged.priority

TaggedCollectionWrapper<T,E> implements *TaggedCollection*<T,E>

└ **TaggedPriorityQueueWrapper**<T,E> implements *TaggedPriorityQueue*<T,E>

Internal Representation: The tagged priority queue wrapper can wrap any priority queue implementation, where each element in the priority queue is a tagged element. It is required that the comparator depends only on the tag, and the tag is immutable except through the use of the *update* method.

In order to have access to the tag within a tagged element, the tagged priority queue wrapper introduces a static class *PQTaggedElement* that extends *TaggedElement* so that a *setTag* method can be provided for internal use within the tagged priority queue wrapper.

```
static class PQTaggedElement<T, E> extends TaggedElement<T, E> {
    PQTaggedElement(T t, E e) {
        super(t, e);
    }
    void setTag(T newTag) {
        this.tag = newTag;
    }
}
```

Key								
⦿	Excellent							
◯	Very Good							
⊖	Good							
●	Fair							
•	Method does nothing							
⊗	Unsupported method							
Method		TaggedBinaryHeap (not tracked)	TaggedBinaryHeap (tracked)	TaggedLeftistHeap (not threaded)	TaggedLeftistHeap (threaded)	TaggedPairingHeap	TaggedFibonacciHeap	TaggedSkipList (OrderedCollection)
Tagged Priority Queue Methods	clear(), per element	⦿	⦿	⦿	⦿	⦿	⦿	⦿
	contains(e)	●	●	●	●	●	●	⊖
	ensureCapacity(), trimToSize()	●	●	•	•	•	•	•
	extractMax()	◯	◯	◯	◯	◯	◯	⦿
	get(tag), getLocator(tag)	●	●	●	●	●	●	⊖
	getTaggedElements(), per element	⦿	⦿	⦿	⦿	⦿	⦿	⦿
	max()	⦿	⦿	⦿	⦿	⦿	⦿	⦿
	merge(tpq1, tpq2)	●	●	◯	◯	◯	⦿	●
	put(tag, d)	◯	◯	◯	◯	◯	⦿	⊖
	putAll(tc), per element in tc	⦿	⦿	◯	◯	◯	⦿	⊖
	putTracked(tag, d)	◯	◯	◯	◯	◯	⦿	⊖
	remove(tag)	●	●	●	●	●	●	⊖
Other Issues	toString(), per element	⦿	⦿	⦿	⦿	⦿	⦿	⦿
	updateTag(tag, pqloc) - increase priority	◯	◯	◯	◯	◯	⦿	⊖
	updateTag(tag, pqloc) - decrease priority	◯	◯	◯	◯	◯	◯	⊖
	accept(v), per element	⦿	⦿	⦿	⦿	⦿	⦿	⦿
	space usage	⦿	◯	⊖	⊖	⊖	●	◯
	very small constant hidden in O()	✓	✓	✓	✓			
Locator Methods	amortized*					✓	✓	
	randomized							✓
	advance(), get()	⦿	⦿	⊗	⦿	⦿	⦿	⦿
	hasNext(), next()	⦿	⦿	◯	⦿	⦿	⦿	⦿
	remove()	◯	◯	◯	◯	◯	◯	⦿
	retreat()	⦿	⦿	⊗	⦿	⦿	⦿	⦿
	update(o) - increase priority	◯	◯	◯	◯	◯	⦿	⊖
	update(o) - decrease priority	◯	◯	◯	◯	◯	◯	⊖
Time Complexity								
⦿	O(1) time							
◯	O(log n) worst case							
⊖	O(logn) expected time							
●	O(n) time							
Space Usage								
⦿	≈ n to 2n							
◯	≈ 3n to 4n							
⊖	≈ 5n to 7n							
●	≈ 10n							

Table 49.3 Trade-offs among the data structures that implement the TaggedPriorityQueue ADT.

```
    }
}
```

Methods

Any method that returns a locator is overridden to return a priority queue locator. The other methods introduced are specific to a tagged priority queue.

```
public TaggedPriorityQueueWrapper(Collection<TaggedElement<T, E>> pairs) {
    super(pairs);
}
```

The *max* method returns a tagged element with a highest priority tag. It throws a *NoSuchElementException* when this collection is empty.

```
public TaggedElement<T, E> max() {
    return ((PriorityQueue<TaggedElement<T, E>>) pairs).max();
}
```

The *getLocator* method takes *tag*, the target tag. It returns a priority queue locator that has been placed at a tagged element with an equivalent tag. It throws a *NoSuchElementException* when there is no element with an equivalent tag.

```
public PriorityQueueLocator<TaggedElement<T, E>> getLocator(T tag) {
    TaggedElement<T, E> entry = pairs.getEquivalentElement(target.setTag(tag));
    return (PriorityQueueLocator<TaggedElement<T, E>>) pairs.getLocator(entry);
}
```

The *put* method takes *tag*, the tag for the tagged element to add, and *data*, the associated data. It creates a new tagged element with the given values, and inserts this tagged element into this collection. It throws an *AtCapacityException* when the collection is already at capacity.

```
public synchronized void put(T tag, E data) {
    pairs.add(new PQTaggedElement<T, E>(tag, data));
}
```

The *putTracked* method takes *tag*, the tag for the tagged element to add, and *data*, the associated data. It creates a new tagged element with the given values, and inserts this tagged element into this collection, and returns a priority queue locator that tracks the new tagged element. It throws an *AtCapacityException* when the collection is already at capacity.

```
public synchronized PriorityQueueLocator<TaggedElement<T, E>> putTracked(T tag,
                                                                    E data) {
    return (PriorityQueueLocator<TaggedElement<T, E>>)
        ((Tracked<TaggedElement<T, E>>) pairs).addTracked(
            new PQTaggedElement<T, E>(tag, data));
}
```

The *extractMax* method removes a tagged element with a highest priority tag, and returns the removed tagged element. It throws a *NoSuchElementException* when this collection is empty.

```
public TaggedElement<T, E> extractMax() {
    return ((PriorityQueue<TaggedElement<T, E>>) pairs).extractMax();
}
```

The *updateTag* method takes *tag*, the new tag, and *loc*, a priority queue locator specifying the tagged element to update. This method replaces the tag of the tagged element at the locator position by *tag*, and makes any required updates in the wrapped priority queue.

```
public void updateTag(T tag, PriorityQueueLocator<TaggedElement<T, E>> loc) {
    PQTaggedElement<T, E> te = (PQTaggedElement<T, E>) loc.get();
    Comparator<? super T> comp = ((TaggedElementComparator<T>)
        ((AbstractCollection) pairs).getComparator()).getTagComparator();
    int comparison = comp.compare(tag, te.getTag());
    te.setTag(tag);
    if (comparison > 0)
        loc.increasePriority(te);
    else if (comparison < 0)
        loc.decreasePriority(te);
}
```

Correctness Highlights: By the correctness of the locator *get* method, *te* is the tagged element at the current locator position. The tag is set, and *increasePriority* is called if the priority of the tag has increased, and *decreasePriority* is called if priority of the tag has decreased. The rest of the correctness follows from that of the locator *increasePriority* and *decreasePriority* method of the wrapped priority queue implementation. ■

The *iterator* method creates and returns new iterator initialized to FORE.

```
public Iterator<TaggedElement<T, E>> iterator() {
    return pairs.iterator();
}
```

49.8.2 Tagged Binary Heap

The only method needed is the constructor. For all tagged priority queues the most general constructor takes *comp*, the comparator that defines the ordering among the tags, and *tracked*, which indicates whether or not a track implementation is desired.

```
public class TaggedBinaryHeap<T, V> extends TaggedPriorityQueueWrapper<T, V>
    implements TaggedPriorityQueue<T, V> {

    public TaggedBinaryHeap(int initialCapacity, Comparator<? super T> comp,
        boolean tracked) {
        super(new BinaryHeap<TaggedElement<T, V>>(initialCapacity,
            new TaggedElementComparator<T>(comp), tracked));
    }
}
```

49.8.3 Tagged Leftist Heap

```
public class TaggedLeftistHeap<T, V> extends TaggedPriorityQueueWrapper<T, V>
    implements TaggedPriorityQueue<T, V> {
```



```

    public TaggedLeftistHeap(Comparator<? super T> comp) {
        super(new LeftistHeap<TaggedElement<T, V>>
            (new TaggedElementComparator<T>(comp)));
    }
}

```

49.8.4 Tagged Pairing Heap

```

public class TaggedPairingHeap<T, V> extends TaggedPriorityQueueWrapper<T, V>
    implements TaggedPriorityQueue<T, V> {

    public TaggedPairingHeap(Comparator<? super T> comp) {
        super(new PairingHeap<TaggedElement<T, V>>
            (new TaggedElementComparator<T>(comp)));
    }
}

```

Since the greedy tree builder of `WeightedGraph` (Chapter 56) creates a tagged pairing heap by calling the no-argument constructor that uses a default comparator, we include that constructor here as an additional example.

```

    public TaggedPairingHeap() {
        this(Objects.DEFAULT_COMPARATOR);
    }
}

```

49.8.5 Tagged Fibonacci Heap

```

public class TaggedFibonacciHeap<T, V> extends TaggedPriorityQueueWrapper<T, V>
    implements TaggedPriorityQueue<T, V> {

    public TaggedFibonacciHeap(Comparator<? super T> comp) {
        super(new FibonacciHeap<TaggedElement<T, V>>
            (new TaggedElementComparator<T>(comp)));
    }
}

```

49.9 Tagged Ordered Collection ADT

package collection.tagged.ordered

```

TaggedCollection<T,E>
    ↑ TaggedOrderedCollection<T,E>

```

In this section we present the `TaggedOrderedCollection` ADT, which is a tagged variation of the `OrderedCollection` ADT. While the comparator provided for an ordered collection might use a single field within the element (that can be viewed as a tag), in a tagged ordered collection there is an explicit association created from the tag to the associated data element. Also the comparator is required to be based upon only the tags. The tags held within a tagged ordered collection need not be unique.

Example Application for a TaggedOrderedCollection

Consider an archive of surveillance images, where each image has a time stamp for when it was taken. Suppose the application needs efficient support to retrieve all images taken between two specified times. If the time stamp is not part of the image itself or would be expensive to extract for each comparison, an association between the time stamp and image is needed. The choice of a tagged ordered collection (versus some other tagged collection) is based on the fact that the application requires the ability to perform a range search over the tags. Finally, an ungrouped tagged collection is more appropriate than a grouped tagged collection when the expectation is that two images would rarely (or never) have the same time stamp.

Interface

The following methods are added to those of the *TaggedCollection* interface.

***TaggedElement*<*T,E*> *min*()**: Returns a tagged element for the least tag in the collection (according to the comparator). More specifically, the first tagged element in the iteration order is returned. This method throws a *NoSuchElementException* when the collection is empty.

***TaggedElement*<*T,E*> *max*()**: Returns a tagged element for the greatest tag in the collection (according to the comparator). More specifically, the last tagged element in the iteration order is returned. This method throws a *NoSuchElementException* when the collection is empty.

***T predecessor*(*T tag*)**: Returns the largest tag used by some tagged element in the collection that is less than *tag*. It does not require that *tag* is in use. It throws a *NoSuchElementException* when there is no tag in the collection less than *tag*.

***T successor*(*T tag*)**: Returns smallest tag used by some tagged element in the collection that is greater than *tag*. It does not require that *tag* is in use. It throws *NoSuchElementException* when there is no tag used in the collection greater than *tag*.

***Locator*<*E*> *iteratorAtEnd*()**: Returns a locator that has been initialized to AFT. As with the *iterator* method, this method enables navigation.

The *iterator* method is inherited from the *Collection* interface. However, here the iteration order is defined by the Comparator *comp*.

Competing ADTs

We briefly discuss ADTs that may be appropriate in a situation when a *TaggedOrderedCollection* ADT is being considered.

OrderedCollection ADT: If the object used for the tag is contained within the associated data element, and it is appropriate to define an comparator based on it, then an *OrderedCollection* should be considered since it slight reduces the space usage. However, it is important to consider whether an *OrderedCollection* ADT or *TaggedOrderedCollection* ADT is more natural for the application.

TaggedBucketOrderedCollection ADT: If each element is identified by tag but there are expected to be multiple elements with some tags, then a *TaggedBucketOrderedCollection* should be considered. The advantage of a tagged bucket ordered collection over a tagged ordered collection is that the complexity of the primary methods depends on the number of unique tags as opposed to the number of tagged elements in the collection.

TaggedDigitizedOrderedCollection ADT: A digitized ordered collection is more appropriate than an ordered collection when (1) the tags can be viewed as a sequence of digits, and (2) performing $O(\log n)$ comparisons between tags is more expensive than extracting the average number of digits in the prefix of a tag needed to distinguish it from the other tags in use. A tagged digitized ordered collection is also indicated when the application requires either a method to efficiently find all elements with a tag beginning with a given prefix, or a method to find the set of elements whose tags have the longest prefix in common with a given tag.

TaggedPriorityQueue ADT: If the ordering defined by the comparator (over the tags) corresponds to a priority for the associated element and the application primarily accesses the highest priority item, and possibly accesses or removes items using a locator, then a tagged priority queue should be considered. Because a tagged priority queue does not need to support a method to efficiently get the element associated with a tag, data structures that implement a tagged priority queue can perform the other methods more efficiently (though asymptotically equivalent) and using less space. However, an tagged ordered collection should be chosen in favor of a tagged priority queue when the required iteration order is based on the ordering of the elements defined by the tags.

Mapping ADT: If the tagged elements held in the collection are comparable, but the application only needs to get the element given its associated tag, put new elements and their associated tag in the collection, and remove the elements associated with a given tag, then a mapping is a better choice since it can perform all of these operations in expected constant time. One would use an tagged ordered collection instead of a mapping when application requires methods that concern an ordering among the elements as defined by the tags.

Selecting a TaggedOrderedCollection Data Structures

The tagged ordered collection wrapper implements all of the *TaggedOrderedCollection* methods in terms of a wrapped ordered collection instance. The process of selecting which ordered collection data structure to wrap follows the process described in Section 29.6 on page 447. [Table 49.4](#) provides a high-level summary of the trade-offs among the various tagged ordered collection data structures.

TaggedSortedArray: The tagged sorted array is very efficient in its use of space, and provides the fastest search time independent of the access pattern. It also supports constant time access to the element with a tag in the given rank. However, adding or removing tagged elements requires shifting elements that follow it. Thus, a tagged sorted array is a good choice only when the tags of the elements added or removed are near the maximum tag, or when there are few mutations after putting all associations into the collection.

TaggedBinarySearchTree: When elements are inserted into a tagged binary search tree in a random order, then it has the desirable property of having expected logarithmic search time for an element with a given tag. However, the tagged binary search tree can degenerate to a sorted list, which has linear search cost.

TaggedRedBlackTree: The tagged red-black tree modifies the tree structure to ensure that at most $2 \log_2 n$ comparisons are made for any search. The tagged red-black tree is the best general-purpose tagged ordered collection data structure since it has guaranteed worst-case logarithmic bounds on the tree height, and has very fast search time. If it is important to have constant time methods to reach the previous or next tagged element in the iteration order, additional structure can be added for this purpose at the expense of increased execution time and space usage.

Key ⦿ Excellent ○ Very Good ● Fair • Method does nothing

Time Complexity

- ⦿

O(1) time
- O(log n) time
- O(n) time
- method does nothing

Space Usage

- ⦿

Excellent
- Very Good
- Good

Table 49.4 Trade-offs among the data structures that implement the TaggedOrderedCollection ADT.

TaggedSplayTree: A tagged splay tree restructures the tree so that the tagged element accessed last is brought to the root in a way that a random sequence of tag accesses leads to a tree with expected logarithmic height. Furthermore, recently accessed tagged elements are near the root, and can therefore be located very efficiently. The tagged splay tree should be used if the goal is to minimize the expected access time, provided that the access pattern is stable and not uniform (i.e., some tags are accessed much more frequently than others), and provided that tags recently accessed are more likely to be accessed again soon.

TaggedBTree: A tagged B-tree is an extension of a balanced tagged binary search tree in which each node can hold between $t - 1$ and $2t - 1$ elements, where integer $t > 1$ is provided as a parameter to the constructor. The tagged B-tree is designed for tagged collections that are so large that secondary storage must be used. In fact, if the elements on secondary storage are very large, it makes sense to use a B-tree or B+-tree as an index, where the tagged element does not hold the internal data, but instead holds only the tag and a disk location.

TaggedBPlusTree: The tagged B+-tree is variation of the tagged B-tree in which the internal nodes are used only for navigation. The leaves (from left to right) form a sorted list of all tagged elements in the collection. The tagged B+-tree is also designed for tagged collections so large that secondary storage is required. It aims to minimize the number of disk access required to both search for elements by the tag, and to find the set of tagged elements that fall in a desired range of tags. A tagged B+-tree also supports efficient **bulk loading**.

TaggedSkipList: A tagged skip list is a sorted list with additional structure that supports finding an element based on its tag in expected logarithmic time. Unlike balanced tagged search trees, a tagged skip list achieves logarithmic performance without rearranging the structure when tagged elements are added or removed from the collection.

49.9.1 Tagged Ordered Collection Wrapper

package `collection.tagged.ordered`

`TaggedCollectionWrapper<T,E>` implements *TaggedCollection<T,E>*

└ `TaggedOrderedCollectionWrapper<T,E>`
implements *TaggedOrderedCollection<T,E>*, *TrackedTagged<T,E>*

Internal Representation: The tagged ordered collection wrapper can wrap any ordered collection implementation, where each element in the collection is a tagged element. It is required that the comparator depends only on the tag, and the tag is immutable.

Methods

In this section we described the methods in the `TaggedOrderedCollectionWrapper` class.

```
public TaggedOrderedCollectionWrapper(
    OrderedCollection<TaggedElement<T, E>> pairs) {
    super(pairs);
}
```

The *min* method returns a tagged element with the smallest tag in the collection. It throws a *NoSuchElementException* when the collection is empty.

```
public TaggedElement<T, E> min() {
    return ((OrderedCollection<TaggedElement<T, E>>) pairs).min();
}
```

The *max* method returns a tagged element with the largest tag in the collection. It throws a *NoSuchElementException* when the collection is empty.

```
public TaggedElement<T, E> max() {
    return ((OrderedCollection<TaggedElement<T, E>>) pairs).max();
}
```

The *predecessor* method takes *tag*, the tag for which to find the predecessor. It returns the largest tag used by some tagged element in the collection that is less than *tag*. This method does not require that *tag* is in use. It throws a *NoSuchElementException* when there is no tag used in the collection less than *tag*.

```
public T predecessor(T tag) {
    return ((OrderedCollection<TaggedElement<T, E>>)
        pairs).predecessor(target.setTag(tag)).getTag();
}
```

The *successor* method takes *tag*, the tag for which to find the successor. It returns the smallest tag used by some tagged element in the collection that is greater than *tag*. This method does not require that *tag* is in use. It throws a *NoSuchElementException* when there is no tag used in the collection greater than *tag*.

```
public T successor(T tag) {
    return ((OrderedCollection<TaggedElement<T, E>>)
        pairs).successor(target.setTag(tag)).getTag();
}
```

The *putTracked* method takes *tag*, the tag for the tagged element to add, and *element*, the associated data. It creates a new tagged element with the given values and inserts this tagged element into this collection. It returns a locator that tracks the new tagged element. It throws an *AtCapacityException* when the collection is already at capacity.

```
public synchronized Locator<TaggedElement<T, E>> putTracked(T tag, E element) {
    return ((Tracked<TaggedElement<T, E>>) pairs).addTracked(
        new TaggedElement<T, E>(tag, element));
}
```

The *iteratorAtEnd* method creates and returns a new locator initialized to AFT.

```
public Locator<TaggedElement<T, E>> iteratorAtEnd() {
    return ((OrderedCollection<TaggedElement<T, E>>) pairs).iteratorAtEnd();
}
```

49.9.2 Tagged Sorted Array

The only method needed is the constructor. For tagged ordered collections the most general constructor takes *comp*, the comparator that defines the ordering among the tags. If desired, a convenience constructor that takes no parameters and uses a default comparator could be added.

```

public class TaggedSortedArray<T, E> extends TaggedOrderedCollectionWrapper<T, E> {

    public TaggedSortedArray(Comparator<? super T> comp) {
        super(new SortedArray<TaggedElement<T, E>>
            (new TaggedElementComparator<T>(comp)));
    }
}

```

49.9.3 Tagged Binary Search Tree

```

public class TaggedBinarySearchTree<T, E>
    extends TaggedOrderedCollectionWrapper<T, E> {

    public TaggedBinarySearchTree(Comparator<? super T> comp) {
        super(new BinarySearchTree<TaggedElement<T, E>>
            (new TaggedElementComparator<T>(comp)));
    }
}

```

49.9.4 Tagged Splay Tree

```

public class TaggedSplayTree<T, E> extends TaggedOrderedCollectionWrapper<T, E> {

    public TaggedSplayTree(Comparator<? super T> comp) {
        super(new SplayTree<TaggedElement<T, E>>
            (new TaggedElementComparator<T>(comp)));
    }
}

```

49.9.5 Tagged Red-Black Tree

```

public class TaggedRedBlackTree<T, E> extends TaggedOrderedCollectionWrapper<T, E> {

    public TaggedRedBlackTree(Comparator<? super T> comp) {
        super(new RedBlackTree<TaggedElement<T, E>>
            (new TaggedElementComparator<T>(comp)));
    }
}

```

49.9.6 Tagged B-Tree

Since the B-tree implementation does not support removal through a locator, the *remove* method is overridden to first retrieve and save a reference to the tagged element that is to be removed using *getEquivalentElement*, and then remove that element. Finally, the element from within the removed tagged element is returned.

```

public class TaggedBTree<T, E> extends TaggedOrderedCollectionWrapper<T, E> {

    public TaggedBTree(Comparator<? super T> comp, int order) {
        super(new BTree<TaggedElement<T, E>>
            (new TaggedElementComparator<T>(comp), order));
    }
}

```

49.9.7 Tagged B+-Tree

```

public class TaggedBPlusTree<T, E> extends TaggedOrderedCollectionWrapper<T, E> {

    public TaggedBPlusTree(Comparator<? super T> comp, int order) {
        super(new BPlusTree<TaggedElement<T, E>>
            (new TaggedElementComparator<T>(comp), order));
    }

    public synchronized E remove(T tag) {
        TaggedElement<T, E> te = pairs.getEquivalentElement(target.setTag(tag));
        pairs.remove(target);
        return te.getElement();
    }
}

```

49.9.8 Tagged Skip List

```

public class TaggedSkipList<T, E> extends TaggedOrderedCollectionWrapper<T, E> {

    public TaggedSkipList(int size, Comparator<? super T> comp) {
        super(new SkipList<TaggedElement<T, E>>
            (size, new TaggedElementComparator<T>(comp)));
    }
}

```

49.10 Tagged Digitized Ordered Collection ADT

package collection.tagged.ordered

```

TaggedCollection<T,E>
└─ TaggedOrderedCollection<T,E>
    └─ TaggedDigitizedOrderedCollection<T,E>

```

The TaggedDigitizedOrderedCollection ADT is a tagged variation of the DigitizedOrderedCollection ADT. While the digitizer provided for an ordered digitized collection might use a single field

within the element (that can be viewed as a tag), in a tagged digitized ordered collection there is an explicit association created from the tag to the associated data element. Also, the digitizer is required to be based upon only the tags. The tags held within a tagged digitized ordered collection need not be unique.

Example Application for a TaggedDigitizedOrderedCollection

A firewall is an example usage for a tagged digitized ordered collection. In particular, the firewall would maintain a large set of prefixes of IP addresses, where associated with each prefix is information about the allowed access level for all IP addresses with that prefix. The choice of a tagged digitized ordered collection (versus some other tagged collection) is based on the fact that IP addresses are easily digitized and the access permission is specified for a set of IP addresses according to a prefix.

Interface

The following methods are added to those of the *TaggedOrderedCollection* interface.

void completions(T prefix, Collection<? super TaggedElement<T,E>> tc): Adds to the provided tagged collection, *tc*, all tagged elements for which the tag has the given prefix. (We consider a tag to be a prefix of itself.)

void longestCommonPrefix(T tag, Collection<? super TaggedElement<T,E>> tc): Adds to the provided tagged collection, *tc*, all tagged elements in this collection whose tag has a longest common prefix with *tag*.

Competing ADTs

A tagged digitized ordered collection can be used only when the tags can be viewed as a sequence of digits and a digitizer can be defined over the tags so that $a \prec b$ exactly when the application considers *a* to be smaller than *b*. If such a requirement cannot be satisfied, then another ADT is needed. Also, if the *getDigit* method of the digitizer is expensive, yet there is an efficient method to compare two tags, then another ADT should be considered since all data structures that implement a tagged digitized ordered collection make heavy use of the *getDigit* method instead of making direct comparisons. We briefly discuss ADTs that may be appropriate in a situation when a Tagged-DigitizedOrderedCollection ADT is being considered.

DigitizedOrderedCollection ADT: If the object used for the tag is contained within the associated data element, and it is appropriate to define a digitizer based on it, then a Digitized-OrderedCollection should be considered since it slightly reduces the space usage. However, it is important to consider whether a DigitizedOrderedCollection ADT or TaggedDigitized-OrderedCollection ADT is more natural for the application.

TaggedBucketDigitizedOrderedCollection ADT: If each element is identified by tag but there are expected to be multiple elements with some tags, then a TaggedBucketDigitizedOrderedCollection should be considered. The advantage of a tagged bucket digitized ordered collection over a tagged digitized ordered collection is that the complexity of the primary methods depends on the number of unique tags as opposed to the number of tagged elements in the collection.

TaggedOrderedCollection ADT: A tagged ordered collection is more appropriate than a tagged digitized ordered collection when (1) the cost of performing $O(\log n)$ comparisons between

tags is less than the average number of digits in a tag, and (2) the application requires neither a method to efficiently retrieve all elements in the collection with a tag having a given prefix, nor a method to retrieve all elements whose tags have a longest prefix in common with a given tag.

Mapping ADT: If a tagged digitized ordered collection is being used primarily to retrieve the element(s) with a given tag, then the Mapping ADT should be considered instead. This ADT is particularly favorable if a constant number of comparisons between tags is less expensive than performing *getDigit* for the length of the prefix of the desired tag that must typically be examined to distinguish it from the other tags in use. A mapping should also be considered if minimizing space usage is important.

Selecting a TaggedDigitizedOrderedCollection Data Structures

Table 49.5 provides a high-level summary of the trade-off between the data structures we present for the TaggedDigitizedOrderedCollection ADT. For a more complete discussion of the trade-offs between the trie, compact trie, compressed trie, Patricia trie, and ternary search trie implementations, see Section 39.4 on page 622.

TaggedTrie: The tagged trie data structure is the simplest *TaggedDigitizedOrderedCollection* implementation. However, it typically requires significantly more space than the other data structures.

TaggedCompactTrie: The tagged compact trie data structure modifies the tagged trie by replacing any leaf that has no siblings by its parent. This significantly reduces the space usage without any additional cost except that the search, insert, and remove methods are slightly more complicated. The length of the search path in a tagged compact trie is exactly the length of the prefix needed to distinguish the given tag from all other tags used in the collection.

TaggedCompressedTrie: The tagged compressed trie performs additional compression on a tagged compact trie to ensure that all nodes have at least two children. The length of the search path in a tagged compressed trie is at most the length of the prefix of the tag needed to distinguish it from all other tags used in the collection.

TaggedPatriciaTrie: The tagged Patricia trie is a variation of a tagged compressed trie that can be used when the digitizer has base 2, and the collection is naturally prefix-free (without adding an end of string character). The most common application is when the tags are fixed-length binary strings (such as ASCII codes or IP addresses). A tagged Patricia trie reduces the space usage of the tagged compressed trie.

TaggedTernarySearchTrie: The tagged ternary search trie is a hybrid between a tagged trie and a tagged binary search tree that combines the time efficiency of a tagged trie with the space efficiency of a tagged binary search tree (Section 49.9). The search path in a tagged ternary trie is not bounded by the number of digits in the tag, but rather has a logarithmic dependence on the size of the collection.

Key						
⊙	Excellent					
○	Very Good					
⊖	Good					
●	Fair					
•	Method does nothing					
Method		TaggedTrie	TaggedCompactTrie	TaggedCompressedTrie	TaggedPatriciaTrie	TaggedTernarySearchTrie
Tagged Digitized Ordered Collection Methods	clear()	●	●	●	●	●
	completions(prefix) *	○	○	○	○	⊖
	contains(tag)	○	○	○	○	⊖
	ensureCapacity(), trimToSize()	•	•	•	•	•
	gett(tag), getLocator(tag)	○	○	○	○	⊖
	getTaggedElements(), per element	⊙	⊙	⊙	⊙	⊙
	longestPrefixMatches(prefix) *	○	○	○	○	⊖
	max(), min()	○	○	○	○	⊖
	predecessor(tag)	○	○	○	○	⊖
	put(tag, d), putTracked(tag, d)	○	○	○	○	⊖
	putAll(tc), time per element in tc	○	○	○	○	⊖
	remove(tag)	○	○	○	○	⊖
	successor(tag)	○	○	○	○	⊖
Other Issues	accept(v), toString()	●	●	●	●	●
	trimToSize()	•	•	•	•	•
	typical space	●	⊖	⊖	⊙	○
Locator Methods	requires b=2 (cannot add end of string character)				✓	
	structure uniquely defined by collection	✓	✓	✓		
	advance(), get(), hasNext(), next()	⊙	⊙	⊙	⊙	⊙
	remove()	○	○	○	○	⊖
	retreat()	⊙	⊙	⊙	⊙	⊙

Time Complexity

- ⊙ O(1) time
- O(d) or O(d_{max}) time
- ⊖ O(d log b) expected time
- O(n) time
- * additional O(|x|) time where x is returned

Expected Space Usage

- ⊙ ≈ 2n (only applies when b = 2)
- ≈ 3n(1 + 1/ln b) ≈ 4n when b = 27
- ⊖ ≈ bn / ln b ≈ 8n when b = 27
- > bn(1 + 1/ln b) ≈ 35n when b = 27

Table 49.5 Trade-offs among the data structures that implement the TaggedDigitizedOrdered-Collection ADT.

49.10.1 Tagged Digitized Ordered Collection Wrapper

package collection.tagged.ordered.digitized

TaggedCollectionWrapper<T,E> implements *TaggedCollection*<T,E>

└ TaggedOrderedCollectionWrapper<T,E> implements *TaggedPriorityQueue*<T,E>

└ TaggedDigitizedOrderedCollectionWrapper<T,E>

implements *TaggedOrderedCollection*<T,E>, *TrackedTagged*<T,E>

Internal Representation: The tagged ordered collection wrapper can wrap any digitized ordered collection implementation, where each element in the collection is a tagged element. It is required that the comparator depends only on the tag, and the tag is immutable.

Methods

We assume that the collection instance provided for the constructor is created within the constructor with no referenced saved by the calling program to prevent the user from accessing the wrapped collection except through the TaggedOrderedCollectionWrapper methods.

```
public TaggedDigitizedOrderedCollectionWrapper
    (DigitizedOrderedCollection<TaggedElement<T, E>> pairs) {
    super(pairs);
}
```

The *completions* method takes *prefix*, the desired prefix, and *tc*, the tagged collection to append all tagged elements for which the tag has the given prefix. Recall that we consider a tag to be a prefix of itself.

```
public void completions(T prefix, Collection<? super TaggedElement<T, E>> tc) {
    ((DigitizedOrderedCollection<TaggedElement<T, E>>)
        pairs).completions(target.setTag(prefix), tc);
}
```

The *longestCommonPrefix* method takes *tag*, the desired prefix, and *tc*, the tagged collection to append all tagged elements in this collection where the tag has a longest common prefix with *tag*.

```
public void longestCommonPrefix(T tag, Collection<? super TaggedElement<T, E>> tc) {
    ((DigitizedOrderedCollection<TaggedElement<T, E>>)
        pairs).longestCommonPrefix(target.setTag(tag), tc);
}
```

49.10.2 Tagged Trie

The only method needed is the constructor. For tagged digitized ordered collections the most general constructor takes *digitizer*, the digitizer to be used to define the digits for any tag, and that implicitly defines the ordering among the tags.

```
public class TaggedTrie<T, E> extends TaggedDigitizedOrderedCollectionWrapper<T, E> {

    public TaggedTrie(Digitizer<? super T> digitizer) {
        super(new Trie<TaggedElement<T, E>>
```

```

        (new TaggedElementDigitizer<T>(digitizer)));
    }
}

```

49.10.3 Tagged Compact Trie

```

public class TaggedCompactTrie<T, E>
    extends TaggedDigitizedOrderedCollectionWrapper<T, E> {

    public TaggedCompactTrie(Digitizer<? super T> digitizer) {
        super(new CompactTrie<TaggedElement<T, E>>
            (new TaggedElementDigitizer<T>(digitizer)));
    }
}

```

49.10.4 Tagged Compressed Trie

```

public class TaggedCompressedTrie<T, E>
    extends TaggedDigitizedOrderedCollectionWrapper<T, E> {

    public TaggedCompressedTrie(Digitizer<? super T> digitizer) {
        super(new CompressedTrie<TaggedElement<T, E>>
            (new TaggedElementDigitizer<T>(digitizer)));
    }
}

```

49.10.5 Tagged Patricia Trie

```

public class TaggedPatriciaTrie<T, E>
    extends TaggedDigitizedOrderedCollectionWrapper<T, E> {

    public TaggedPatriciaTrie(Digitizer<? super T> digitizer) throws Exception {
        super(new PatriciaTrie<TaggedElement<T, E>>
            (new TaggedElementDigitizer<T>(digitizer)));
    }
}

```

49.10.6 Tagged Ternary Search Trie

```

public class TaggedTernarySearchTrie<T, E>
    extends TaggedDigitizedOrderedCollectionWrapper<T, E> {

    public TaggedTernarySearchTrie(Digitizer<? super T> digitizer) {
        super(new TernarySearchTrie<TaggedElement<T, E>>
            (new TaggedElementDigitizer<T>(digitizer)));
    }
}

```

49.11 Tagged Spatial Collection ADT

package collection.tagged.spatial

TaggedCollection<*T*,*E*>

└ *TaggedSpatialCollection*<*T*,*E*>

In this section we present the *TaggedSpatialCollection* ADT, which is a tagged variation of the *SpatialCollection* ADT.

Example Application for a *TaggedSpatialCollection*

An example usage for a tagged spatial collection would be to maintain a set of readings (with longitude, latitude and time) for a mobile device. A query might ask which devices were in a certain geographic region (specified by a longitude and latitude) range during a given time range. The tag would be a three-dimensional data object and the associated data would hold information about the mobile device. Since the location and time are not part of the static information associated with each mobile device, it is natural to treat this information as a tag. The choice of a tagged spatial collection (versus some other tagged collection) is based on the fact that the application requires the ability to find all devices in a three-dimensional box defined by the tags. Finally, an ungrouped tagged collection is preferable to a tagged bucket spatial collection when the expectation is that two devices would rarely (or never) have the same tag (i.e., be at the same location at the same time).

Interface

The following methods are added to those of the *TaggedCollection* interface.

***TaggedElement*<*T*,*E*> min(int dimension):** Returns a tagged element for the least tag in the collection (according to the comparator) along the given dimension. It throws a *NoSuchElementException* when the collection is empty.

***TaggedElement*<*T*,*E*> max(int dimension):** Returns a tagged element for the greatest tag in the collection (according to the comparator) in the given dimension. It throws a *NoSuchElementException* when the collection is empty.

***Collection*<*TaggedElement*<*T*,*E*>> withinBounds(*E* minCorner, *E* maxCorner):** Returns a collection of the tagged elements for which the tag fall within (or on) the boundary of the multidimensional box defined by the two given corners, *minCorner* and *maxCorner*. The method requires that the coordinates of *minCorner* are less than or equal to those of *maxCorner* along every dimension of the spatial collection.

Competing ADTs

We briefly discuss ADTs that may be appropriate in a situation when a *TaggedSpatialCollection* ADT is being considered.

SpatialCollection ADT: If the point used for the tag is contained within the associated data element, and it is appropriate to define an XY comparator or multiple comparators for the different dimensions based on it, then a spatial collection should be considered since it slightly

reduces the space usage. However, it is important to consider whether a `SpatialCollection` ADT or `TaggedSpatialCollection` ADT is more natural for the application.

TaggedBucketSpatialCollection ADT: If each element is identified by tag but there are expected to be multiple elements with the same tag, then a `TaggedBucketSpatialCollection` should be considered. The advantage of a tagged bucket spatial collection over a tagged spatial collection is that the complexity of the primary methods depends on the number of unique tags, as opposed to the number of tagged elements in the collection.

TaggedOrderedCollection ADT: If there is a singly ordering for the tags, then a tagged ordered collection is a better choice.

Selecting a TaggedSpatialCollection Data Structures

See Section 46.4 on page 732 for discussion of the trade-offs between a k -d tree and a quad tree.

Tagged KD-Tree: The tagged k -d tree cycles through the k dimensions at each level of the tree, dividing each region by a halfspace according to the dimensions associated with that level. While the expected height of a tagged k -d tree is roughly d times more than a d -dimensional version of a tagged quad tree, each level only requires a single comparison in a kd tree, versus d comparisons in a d -dimensional version of a tagged quad tree. The space usage of a tagged k -d tree scales well in the number of dimensions, and so it is the tagged spatial collection data structure of choice for spaces larger than 2 or 3 dimensions.

Tagged Quad Tree: A tagged quad tree divides the subspace into four regions (based on the tag) at each internal node. The advantage of the tagged quad tree is that the depth of the tree is smaller than for a tagged k -d tree. While a tagged quad tree can be extended to higher dimensions (e.g., a tagged oct tree is a three-dimensional variant), it does not scale well to higher dimensions since most children for internal nodes will be *null* (corresponding to empty sub regions). Also, deletion for a tagged quad tree is more expensive than for a tagged k -d tree. As a practical matter, the tags for the provided implementation of a tagged quad tree must implement the `XYPoint` interface, whereas the tagged k -d tree implementation allows any tag type provided that a comparator is supplied for each dimension.

49.11.1 Tagged Spatial Collection Wrapper

package `collection.tagged.spatial`

`TaggedCollectionWrapper<T,E>` implements `TaggedCollection<T,E>`

↑ `TaggedSpatialCollectionWrapper<T,E>` implements `TaggedSpatialCollection<T,E>`

Internal Representation: The tagged spatial collection wrapper can wrap any spatial collection implementation, where each element in the collection is a tagged element. It is required that the comparator(s) depend(s) only on the tag, and the tag is immutable.

Instance Variables and Constants: Recall that in the tagged collection wrapper, to avoid the need to create a new tagged element for each search for a target element (as specified by its tag), the global variable *target* is used each time. So that *target* can be used, it is a mutable tagged element (see [page 787](#)). Since a target region is specified by two corners (each a d -dimensional point), we use a second mutable tagged element *anchor* to specify the second corner of a bounding box in an orthogonal range query. In a multithreaded application, Any method that depends upon *target* or *anchor* must be synchronized to ensure that no method call by another thread changes either value before it is used.

```
protected MutableTaggedElement<T, E> anchor = new MutableTaggedElement<T, E>();
```

Methods

We assume that the collection instance provided for the constructor is created within the constructor with no referenced saved by the calling program to prevent the user from accessing the wrapped collection except through the `TaggedSpatialCollectionWrapper` methods.

```
public TaggedSpatialCollectionWrapper(Collection<TaggedElement<T, E>> pairs) {
    super(pairs);
}
```

The *min* method returns a tagged element associated with the largest tag in the collection. It throws a *NoSuchElementException* when the collection is empty.

```
public TaggedElement<T, E> min(int dimension) {
    return ((SpatialCollection<TaggedElement<T, E>>) pairs).min(dimension);
}
```

The *max* method returns a tagged element associated with the largest tag in the collection. It throws a *NoSuchElementException* when the collection is empty.

```
public TaggedElement<T, E> max(int dimension) {
    return ((SpatialCollection<TaggedElement<T, E>>) pairs).max(dimension);
}
```

The *withinBounds* method takes *minCorner*, the bottom left corner of the target box, and *maxCorner*, the upper right corner of the target box. It returns a collection of all tagged elements for which the tag is within (or on) the target box. The method requires that the coordinates of *minCorner* are less than or equal to those of *maxCorner* along every dimension of the spatial collection.

```
public Collection<TaggedElement<T, E>> withinBounds(T minCorner, T maxCorner) {
    return ((SpatialCollection<TaggedElement<T, E>>)
        pairs).withinBounds(target.setTag(minCorner), anchor.setTag(maxCorner));
}
```

49.11.2 Tagged KD-Tree

The only methods needed are the constructor, and a method to wrap the comparators for the *d* dimensions. For a tagged *k*-d tree, the constructor takes *comparators*, an array of *d* comparators for each of the *d* orderings.

```
public class TaggedKDTree<T, E> extends TaggedSpatialCollectionWrapper<T, E> {

    public TaggedKDTree(Comparator<? super T> ...comparators) {
        super(new KDTree<TaggedElement<T, E>>(wrap(comparators)));
    }
}
```

Each comparator of type *T* that is passed to the tagged *k*-d tree must be wrapped so that comparisons of tagged elements are made on the basis of the tag only. The static method *wrap* takes as its parameter *comparators*, an array of comparators of type *T* and returns an array of tagged element comparators that wrap the provided comparators.


```

static <T> Comparator<? super TaggedElement<T, ?>>[]
    wrap(Comparator<? super T>[] comparators) {
    Comparator<? super TaggedElement<T, ?>>[]
        wrapped = new Comparator[comparators.length];
    int i = 0;
    for (Comparator<? super T> comp : comparators)
        wrapped[i++] = new TaggedElementComparator<T>(comp);
    return wrapped;
    }
}

```

49.11.3 Tagged Quad Tree

The only method needed here is the constructor. For a tagged quad tree, the constructor takes *comp*, the XY comparator that defines the two orderings among the tags.

```

public class TaggedQuadTree<T extends XYPoint, E> extends
    TaggedSpatialCollectionWrapper<T, E> {

    public TaggedQuadTree(XYComparator<? super T> comp) {
        super(new QuadTree<TaggedElement<T, E>>
            (new TaggedElementXYComparator<T>(comp)));
    }
}

```

We also provide a convenience constructor that takes no parameter and uses a default XY comparator.

```

public TaggedQuadTree() {
    this(new DefaultXYComparator());
}

```

Chapter 50

Tagged Bucket Collection ADTs

package collection.tagged

Iterable<*TaggedElement*<*T*, *Collection*<*E*>>>

└ *TaggedBucketCollection*<*T*,*E*>

As with the tagged collections, the elements are organized based on an application-specified tag. However, in a tagged bucket collection, each tag maps a set of elements called a bucket. The semantics of *put* is modified to put a new bucket into the collection when the given tag is not yet in use. The new bucket is created by a bucket factory, which gives the application the flexibility to specify the collection type of the bucket. This can be done either by providing a bucket factory to the constructor, or by providing the constructor with a class to be instantiated when a new bucket is required. When the given tag is already in use, then the element is simply added to the existing bucket. Likewise, when an element is removed from a bucket of size 1, then the tag is also removed. In a tagged collection, the cost associated with locating a desired element generally depends on the number of tagged elements in the collection, whereas the cost associated with locating a desired element generally depends on the number of unique tags.

A tagged bucket collection efficiently supports the query of returning the set of all data elements with an associated tag. For example, in an adjacency list representation of a graph (Chapter 55) a set of all edges associated with a vertex is maintained. The adjacency list implementation uses a tagged bucket collection, where the tag is the vertex and the bucket holds edges incident to that vertex. As an example application, consider the problem of indexing a document to efficiently locate all occurrences of a given word in the document. A natural approach is to create tagged elements in which the tag is the word in the document, and the associated data is the offset into the document for an occurrence of the word. In a tagged bucket collection, all occurrences of a given word would be grouped into a single bucket for easy access. A very similar use of a tagged bucket collection for indexing all occurrences of a fixed length sequence of bases in a genome is discussed in the next section, and in the case study in Section 50.2.

Uses: Any tagged collection (Chapter 49) data structure can be wrapped. It also uses the data structure created by the bucket factory (Section 50.3).

Strengths: Allows an application program to efficiently access an element based on its associated tag when there can be more than one element associated with the tag. A tagged bucket collection can wrap any tagged collection data structure to fit the particular needs of the application. An important advantage of using a tagged bucket collection is that cost of the methods are based on the number of distinct tags, as opposed to the number of tagged elements as in a tagged collection.

Weaknesses: If there is only one element associated with a given tag, then some overhead occurs both in the time and space complexity. For such situations the corresponding tagged collection is a better option.

50.1 Case Study: Historical Event Collection (Indexing)

Section 1.5 introduced a case study in which the task was to maintain a collection of historical event objects, where each event object includes a date, a description, and possibly other information. Section 29.1 discussed the ADT and data structure selection to efficiently support the following operations:

1. Insert a new historical event into the collection.
2. Given a date, return a list of all events in the collection that occurred on that date.
3. Given a date, remove all events with that date.
4. Given a date and description, remove all events with the given date and description.
5. Given a date d and a range r , return a list of all events in the collection that occurred in the interval $[d - r, d + r]$.

The OrderedCollection ADT was selected, where the comparator would order the elements by date. We note that since a skip list supports a range search it would be a good choice for a data structure. However, if the number of dates in each range is relatively small and the search time should be minimized, then a red-black tree is most likely a better choice. We will assume that for this application, the goal is to minimize search time, and furthermore that the number of dates in each range is typically small. Therefore, we will continue assuming that a red-black tree is selected.

In this section, we consider what additional data structure(s) are needed to support the following additional operation:

- Given a word, return a list of all events in the collection that include the given word in the event description.

For this operation, an ADT that organizes the events based on a word contained within the description is needed. Furthermore, while one could define a lexicographic ordering among words, no method requires using this ordering. So it suffices to select an ADT in which decisions are based only on whether two words are equivalent. Since one would expect that the same word occurs in many events, a tagged bucket collection (Chapter 50) seems most appropriate where a word is used as a tag. More specifically let $e_1, \dots, e_k$ be historical events that all include the word w in their description. When event e_i is inserted into the collection, it should be inserted into the ordered collection (where the comparator provided is based solely on the date). In addition, an association between w (the tag) and e_i (the associated data) should be added to a tagged bucket collection. The semantics of a tagged bucket collection is that an association is created between w and the bucket $\{e_1, \dots, e_k\}$ where any desired data structure can be used for the bucket. Let n be the number of distinct words that occur among all events in the collection.

Two design decisions remain: (1) How the buckets should be organized with respect to the tag, and (2) what data structure should be used for each bucket. We now consider the first of these decisions. Since the tag (a word) is being used solely as an identifier for the bucket of events whose events contain that word, the BucketMapping ADT is the natural choice. There expected cost to locate the bucket associated with a word is constant. However, part of this cost is the computation of the hashcode for the target word. While this is often viewed as a constant cost, it really depends upon the length (i.e., number of digits) in the word. Let p be the expected length of the prefix needed to distinguish the target word from the n words held in the collection. If the cost to extract these p digits is less than the cost of computing the hash code for w , then a TaggedBucketDigitizedOrderedCollection ADT should be considered. For the sake of this case study, we will assume that the words

are relatively short, so a BucketMapping ADT is selected. Assuming that it is fairly uncommon for a word to be removed from the bucket mapping (which would occur only if the only remaining events containing a given word is removed), an OpenAddressingMapping is the best choice.

We now consider what data structure to use within each bucket. To answer this question, it is important to think the most natural way to return the events to the user. For example, one option for each bucket is to use a positional collection that would maintain the events in the order in which they were inserted into the collection. Here is a sample output for a simple historical event selection where the target word is “pi”:

```
1596: Ludolf van Ceulen computes pi to twenty decimal places
1949: John von Neumann computes pi to 2,037 decimal places using ENIAC
450: Tsu Ch'ung-Chih and Tsu Keng-Chih compute pi to six decimal places
1961: Daniel Shanks and John Wrench compute pi to 100,000 decimal places
1424: Ghiyith al-Kashi computes pi to sixteen decimal places
1987: Yasumasa Kanada et al. compute pi to 134 million decimal places
1706: John Machin computes pi to 100 decimal places
1882: Ferdinand Lindeman proves that pi is transcendental
-260: Archimedes computes pi to two decimal places
```

As a user, it is probably frustrating that the above events do not appear in chronological order. Seeing the events chronologically, as shown below, supports the user in comprehension of the data (in this case, to see progress over time).

```
-260: Archimedes computes pi to two decimal places
450: Tsu Ch'ung-Chih and Tsu Keng-Chih compute pi to six decimal places
1424: Ghiyith al-Kashi computes pi to sixteen decimal places
1596: Ludolf van Ceulen computes pi to twenty decimal places
1706: John Machin computes pi to 100 decimal places
1882: Ferdinand Lindeman proves that pi is transcendental
1949: John von Neumann computes pi to 2,037 decimal places using ENIAC
1961: Daniel Shanks and John Wrench compute pi to 100,000 decimal places
1987: Yasumasa Kanada et al. compute pi to 134 million decimal places
```

Thus, the ADT for each bucket should have an iteration order that is sorted by data. Thus, an ordered collection (with the comparator defined by the date) is the best choice. Since a skip list allows very efficient iteration, it would be a good choice for the bucket.

To summarize, the following support classes and data structures were used in this case study.

- A comparator *eventComp* must be defined that can compare event instances according to their dates. In particular, event *i* should precede event *j* in the ordering exactly when event *i* occurred chronologically before event *j*.
- All events are inserted into a red-black tree that uses *eventComp*. This data structure is used to retrieve an event that occurred on a specified date, or the set of events that occurred in a range of dates. If it is expected that there may be multiple events that occur on the same date, then a tagged red-black tree would be a better choice.
- All events are also inserted into a tagged bucket collection once for every distinct word that occurs in the event description. In particular if event *e* has distinct words $w_1, \dots, w_k$, then *put*(*e*, w_i) is called for $1 \leq i \leq k$. This tagged bucket collection is used to retrieve all events with a specified word.
- The tagged bucket collection will wrap the OpenAddressingMapping class, to allow expected constant time access to the bucket associated with a specified word.

- Finally, each bucket is a skip list so that the iteration order will be chronological, and iteration will be very efficient.

50.2 Case Study: Biosequence Comparison

We illustrate an application of a tagged bucket collection, namely a bucket mapping, in the area of computational biology to support a tool for finding matching substrings in genomic DNA sequences. The same algorithm could be applied to find a desired substring in a text document. This algorithm for biosequence comparison is an important part of well-known tools such as FASTA [122] and BLAST [9, 10].

Background

A DNA sequence is a string of characters, called **bases**, from the alphabet $\{a, c, g, t\}$. Genomic DNA encodes a large collection of features, including:

- genes** - the instructions for building proteins;
- regulatory sites** - sequence markers recognized by cellular machinery that can increase or decrease the rate at which a given gene is used to make protein;
- repeats** - bases left behind by **transposable elements**, pieces of DNA that can autonomously copy themselves and move around in the genome. Transposable elements proliferate, then die, leaving behind many inactive copies of themselves in the genome as repeats.

All DNA is subject to **mutations** that alter its sequence over time. Due to natural selection, functional sequence features like genes and regulatory sites are more resistant to mutation over generations than DNA that does not code for anything because organisms with too many mutations in these features usually do not survive. We can therefore find these features by comparing DNA sequences from two different organisms and seeing which parts of the sequences have remained similar to each other since their lineages split from their last common ancestor.

Abstractly, we are given two long strings s_1 and s_2 of characters, and we want to find short substrings (parts of the features) common to s_1 and s_2 . In general, the common substrings might not be exactly the same because even functional sequences mutate over time; however, we often find that s_1 and s_2 still exhibit exactly matching substrings of at least 10 to 15 characters in the neighborhood of their shared features. By detecting these exact substring matches, we can find the most likely locations of the shared features in s_1 and s_2 and can then apply more sensitive but more expensive similarity search algorithms only at those locations.

The Algorithm

Naively, we could find **k -mers** (i.e., substrings of length k) common to s_1 and s_2 by comparing every k -mer from one sequence to every k -mer from the other. Such an approach would take time $\Theta(|s_1| \cdot |s_2|)$, which is unacceptable because interesting DNA sequences range from thousands to billions of characters in length. The human genome is about three billion characters long.

Fortunately, there is a much better approach. Call s_1 the **corpus string** and s_2 the **pattern string**, and assume we are searching for common substrings of length k . First iterate through each k -mer in the pattern string, putting each k -mer encountered into a bucket mapping using the k -mer as the tag, and its offset in the pattern as the associated data. Note that if a k -mer appears more than once

in a pattern, then all of its offset indices will appear in the same bucket. All matches between the pattern string and corpus string can be found by iterating through all k -mers in the corpus, and if that k -mer is in the bucket mapping, then we have a match between the pattern and corpus. Since a bucket mapping can support insertion and search in expected constant time (e.g., using an open addressing mapping), the overall time complexity to find all matches between the pattern and corpus is $\Theta(|s_1| + |s_2| + M)$ where M is the number of common substrings actually found. In general, this time cost is much lower than the cost of the naive algorithm. Finally, the user could specify a **mask string** that contains “uninteresting” DNA. Any k -mer appearing in the mask string is removed from the bucket mapping before searching for matches in the corpus. Fast substring matching based on fast set implementations (which are wrapped within the bucket mapping) form the core of many of today’s high-speed biosequence matching algorithms.

50.3 Bucket Factory Interface

A bucket factory is used by a tagged bucket collection to create a new bucket for each newly added tag. Using the **abstract factory** design pattern (Section C.1) provides the flexibility to create tagged bucket collections that use a variety of collection types as buckets.

```
public interface BucketFactory<E> {
    public Collection<E> createBucket();
}
```

50.4 Tagged Bucket Collection Interface

We now describe the *TaggedBucketCollection* interface.

TaggedBucketCollection(CollectionFactory<E> factory): Creates a new empty tagged bucket collection with the given collection factory. The factory is used, on demand, to create a bucket for each newly added tag.

void accept(Visitor<TaggedElement<T, Collection<E>>> v): Traverses each element of this collection, in the iteration order, on behalf of the visitor. The visitor sees a tagged element for each tag and its associated bucket.

void clear(): Removes all tagged elements from this tagged bucket collection.

boolean contains(T tag): Returns *true* if an equivalent tag exists in this tagged bucket collection. Otherwise, *false* is returned.

void ensureCapacity(int capacity): Increases the capacity of this tagged collection, if necessary, to ensure that it can hold at least *capacity* distinct tags.

Collection<E> get(T tag): Returns a collection of all elements with the given tag. This method requires that tag be in use and will throw a *NoSuchElementException* otherwise.

Locator<E> getElements(T tag): Returns a locator positioned at FORE within the bucket associated with the given tag, or an iterator over an empty bucket if the tag is not in use.

int getCapacity(): Returns the current capacity (in terms of the number of tags) of this tagged collection.

int getSize(): Returns the number of elements, n , in this collection.

int getNumTags(): Returns the number of distinct tags used in this collection.

int bucketSize(T tag): Returns the number of elements with the given tag.

boolean isEmpty(): Returns *true* if this collection contains no elements, and otherwise returns *false*.

Iterator<TaggedElement<T, Collection<E>>> iterator(): Returns an iterator initialized at *FORE* that may be used to iterate over the tags and their associated buckets.

void put(T tag, E element): This method creates a new tagged element with the given values and inserts this tagged element into this collection. If there is already an element with an equivalent tag, then *element* is added to the bucket. Otherwise, a new bucket is created for *tag*, and *element* is placed in that bucket. An *AtCapacityException*, an unchecked exception, is thrown if the collection is already at capacity.

E remove(T tag): Removes from this tagged collection an arbitrary element with the given tag. It returns the element that was removed, and throws a *NoSuchElementException* when there is no element with an equivalent tag.

boolean remove(T tag, E element): Removes from this tagged collection an element equivalent to the given element with the given tag. It returns *true* if an equivalent element with the given tag was found, and otherwise returns *false*.

boolean removeBucket(T tag): Removes and returns the bucket associated with the given tag.

String toString(): Returns a string that describes each tagged element in the collection, as produced by the *toString* method for that element, in the iteration order.

void trimToSize(): Trims the capacity of this tagged bucket collection to be its current size. An application can use this operation to minimize the space usage. For elastic implementations, this method does nothing.

We briefly discuss other ADTs that may be appropriate in a situation when an tagged bucket collection is being considered.

Collection: If the property used to organize the elements is directly defined over the elements themselves and it will be rare for two elements to have the same tag, then the corresponding collection may be more appropriate than a tagged bucket collection.

Another reason to duplicate the tag that is part of the associated element, is when each element is very large yet the tag is relatively small. If the collection itself is large, then each element will most likely be stored in secondary storage. In such a situation, the introduction of the tagged element, in which the tag is associated with a location on secondary storage for its associated element, enables a more efficient search since it allows a disk page to provide access to as many elements as possible. However, if it will be rare for two elements to have the same tag, then the corresponding tagged collection is a better choice.

Tagged Collection: Once a decision has been made to associate a tag with each element, the remaining question is whether or not the elements that share a tag should be grouped together. The advantage of grouping elements with a shared tag is that the search cost depends on the number of unique tags, versus the number of tagged elements, and also grouping the elements enables efficient access to all elements with a given tag. When it is expected that only a small fraction of elements will share a tag, an ungrouped tagged collection is a better choice.

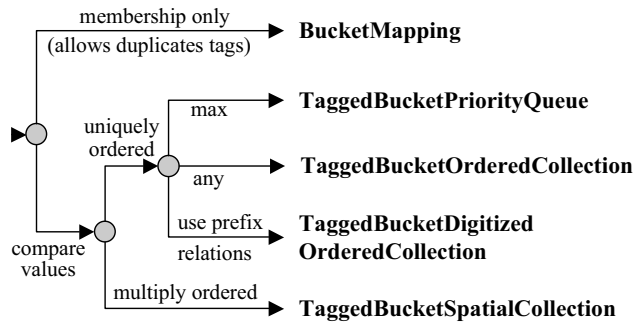


Figure 50.1

A decision tree for selecting an tagged bucket collection ADT.

50.5 Selecting a Tagged Bucket Collection ADT

Once it has been determined that a tagged bucket collection is warranted, the decision process for deciding on a specific tagged bucket collection ADT should be applied as discussed in Section 2.6.2 on page 26. Figure 50.1 provides a decision tree to guide the process of selecting among the tagged bucket collection ADTs discussed in this book.

This section summarizes the purpose of each of the ADTs that implement the tagged bucket collection interface. Each tagged bucket collection is a variation of the corresponding tagged collection in which all elements associated with a given tag are grouped together in a bucket. There are two advantages of a tagged bucket collection over the corresponding tagged collection: (1) to improve the efficiency of the primary methods, since they will depend on the number of unique tags, as opposed to the number of tagged elements in the collection, and (2) to provide efficient support of returning all elements associated with a specified tag or set of tags.

BucketMapping - This ADT is the tagged bucket version of the Mapping ADT. All elements with the same tag are grouped together in a bucket that can be quickly accessed via the tag. As with a mapping, the tag need not be comparable.

TaggedBucketPriorityQueue - The tags must be comparable, and their order is intended to be based on some intrinsic priority among the tags where $t_1 > t_2$ implies that elements in the bucket associated with t_1 have a higher priority than elements in the bucket associated with t_2 . While there is a total order defined over the tags, iteration need not follow this order. If efficient access of the bucket associated with the lowest priority tag is instead desired, a reverse comparator (Section 5.2.3) can be used to wrap the desired comparator.

TaggedBucketOrderedCollection - This ADT is designed for applications for which the tags associated with the buckets are uniquely ordered, and for which iteration must correspond to this unique order. This ADT is well suited for applications that must efficiently find all buckets with tags in a certain range, or find the elements in the iteration order associated with a tag closest to a desired value. For example, in the historical event collection discussed in Sections 1.5 and revisited in Section 29.1, if there are many historical events that occur on the same date, and the application wants to efficiently find the set of all events within a given range of dates, then a tagged bucket ordered collection is the most appropriate ADT.

TaggedBucketDigitizedOrderedCollection - This ADT is a specialized TaggedBucketOrderedCollection ADT in which a digitizer (Section 5.3) is provided for the tags. A total order over

Tagged Bucket Collection ADT	Corresponding Tagged Collection	Section	Page
BucketMapping	Mapping	49.7	797
TaggedBucketPriorityQueue	TaggedPriorityQueue	49.8	801
TaggedBucketOrderedCollection	TaggedOrderedCollection	49.9	807
TaggedBucketDigitizedOrderedCollection	TaggedDigitizedOrderedCollection	49.10	814
TaggedBucketSpatialCollection	TaggedSpatialCollection	49.11	820

Table 50.2 For each tagged bucket collection ADTs, this table lists corresponding tagged collection ADT, the section, and the page where the decision process for it is discussed.

the tags is implicitly defined by the digitizer. This ADT supports all the methods of the TaggedBucketOrderedCollection ADT. In addition, it provides methods to (1) retrieve the set of elements whose tag shares the longest common prefix within all tags in use, and (2) to find all elements whose tag has a given prefix.

TaggedBucketSpatialCollection - This ADT is designed for when the tags associated with the buckets are multiply ordered, and the application must efficiently find the set of elements with a tag within a certain range in each ordering.

50.6 Selecting a Data Structure

The process of selecting a data structure for a tagged bucket collection exactly parallels the decision process for the corresponding tagged collection ADT. Table 50.6 lists the tagged bucket collection ADTs along with the section number and page number for where the corresponding tagged collection is discussed. The rest of the selection process involves selecting a Collection data structure for the bucket. Section 7.3 describes the decision process of selecting an ADT for a collection. Then the chapter on ADT provides guidance on selecting the data structure. For example, in the adjacency list representation for a graph (Chapter 55), the outgoing edges from each vertex are stored in a tagged bucket collection *outEdges* that uses wraps an open addressing mapping, and uses a doubly linked list for the buckets. The TaggedBucketCollectionWrapper (Section 50.7) is allocated using the following statement:

```
TaggedBucketCollection<V, E> outEdges = new TaggedBucketCollectionWrapper<V, E>
    (new OpenAddressingMapping<V, Collection<E>>(), DoublyLinkedList.class);
```

50.7 Tagged Bucket Collection Wrapper

package collection.mapping

TaggedBucketCollectionWrapper<T,E> implements *TaggedBucketCollection<T,E>*

The tagged bucket collection wrapper can be used to create a tagged bucket collection for any tagged collection type.

Internal Representation: The tagged bucket wrapper is implemented as a wrapper for any tagged collection, where each tag is placed in the wrapped collection only once, and the associated element for each tag is a bucket object of any collection type. An important distinction is that the size of a wrapped tagged collection only depends upon the number of distinct tags. Thus, for the time complexity of all methods delegated to the wrapped tagged collection, n is the number of distinct tags. However, from the view of the application, the size of a tagged bucket collection is the number of elements, not the number of tags. For example suppose that a tagged bucket collection includes the associations $\{t_1 \rightarrow B_1, \dots, t_k \rightarrow B_k\}$ where t_i is a tag and B_i is the corresponding bucket. Then the number of elements in the wrapped tagged collection is k , and the number of elements in the tagged bucket collection is $\sum_{i=1}^k |B_i|$ where $|B_i|$ is the number of elements held in bucket B_i .

Instance Variables and Constants: The instance variable *tc* is the wrapped tagged collection that associates each tag with a bucket, and *factory* is the bucket factory that is used to create a new bucket for each tag. Finally, *emptyLocator* is a locator initialized to the front of an empty bucket to return when a locator is requested for an unused tag. There is no instance variable to hold the number of elements in the tagged bucket collection. Instead, the size must be computed as the sum of the sizes of the buckets.

```
protected TaggedCollection<T, Collection<E>> tc;
protected BucketFactory<E> factory;
final Locator<E> emptyLocator;
```

Representation Properties: Since the primary data structure used is the wrapped tagged collection, most of the correctness follows from the correctness of the wrapped implementation. We introduce one property.

NUMELEMENTS: The size of the collection is equal to the total number of elements held in the buckets.

Methods

In this section we present internal and public methods for the TaggedBucketCollectionWrapper class. The representation is exposed so that any type of tagged collection can be provided to suit the performance needs of the application. However, this requires that the provided tagged collection is not externally modified. To prevent accidental modification, it is recommended that a reference to the provided tagged collection not be retained by the user. This prevents the user from accessing the wrapped collection except through the TaggedBucketCollectionWrapper methods. If this risk of exposing the internal representation is problematic, then this implementation could be modified to create a copy of the provided tagged collection to be used internally, at the expense of the time to create the copy.

The main constructor for the tagged bucket collection wrapper takes *tc*, the (empty) tagged collection from T to buckets that is to be wrapped and *factory*, a bucket factory that is used to create a bucket for each new tag.

```
public TaggedBucketCollectionWrapper(
    TaggedCollection<T, Collection<E>> tc, BucketFactory<E> factory){
    if (!tc.isEmpty())
        throw new IllegalArgumentException("The given tagged collection must be empty.");
    this.tc = (TaggedCollection<T, Collection<E>>) tc;
```

```

    this.factory = factory;
    emptyLocator = factory.createBucket().iterator();
}

```

A second constructor takes *tc*, the tagged collection from T to buckets that is to be wrapped and *bucketType*, a class of the appropriate collection type for the buckets. This constructor creates a factory that uses reflection to call the no-argument constructor for the provided bucket type. The method requires the provided class *bucketType* implement the *Collection* interface.

```

public TaggedBucketCollectionWrapper(
    TaggedCollection<T, Collection<E>> tc, final Class bucketType){
    this(tc, new BucketFactory<E>() {
        public Collection<E> createBucket() {
            try {
                return (Collection<E>) bucketType.newInstance();
            } catch (Exception e) {
                throw new RuntimeException("bucket creation failed");
            }
        }
    });
    if (!Collection.class.isAssignableFrom(bucketType))
        throw new IllegalArgumentException("bucket type does not implement Collection");
}

```

TrivialAccessors

The *getTaggedCollection* method returns the underlying tagged collection used by this wrapper.

```

public TaggedCollection<T, Collection<E>> getTaggedCollection() {
    return tc;
}

```

The *getNumTags* method returns the number of tags held in the mapping.

```

public int getNumTags(){
    return tc.getSize();
}

```

Correctness Highlights: Since all elements with the same tag, are stored in the bucket associated with a single tagged element (from the tag to the bucket), the number of distinct tags is the size of the wrapped tagged collection. ■

The *isEmpty* method returns *true* if and only if there are no elements in the collection.

```

public boolean isEmpty() {
    for (TaggedElement<T, Collection<E>> te : tc)
        if (te.getElement().getSize() > 0)
            return false;
    return true;
}

```

The *getSize* method returns the number of elements, among all buckets, held in this collection.

```

public int getSize() {
    int size = 0;
    for (TaggedElement<T, Collection<E>> te : tc)
        size += te.getElement().getSize();
    return size;
}

```

The *getCapacity* method returns the capacity (in terms of the number of tags).

```

public int getCapacity() {
    return tc.getCapacity();
}

```

Algorithmic Accessors

The *contains* method takes *tag*, the target tag, and returns *true* if and only if there is a bucket associated with the given tag. The method performs lazy deletion of buckets that have been emptied directly by the application.

```

public boolean contains(T tag) {
    boolean exists = tc.contains(tag);
    if (exists && tc.get(tag).isEmpty()) {
        tc.remove(tag);
        exists = false;
    }
    return exists;
}

```

The *get* method takes *tag*, the target tag, and returns the bucket holding the elements associated with the given tag. It throws a *NoSuchElementException* when *tag* is not in use. The method performs lazy deletion of buckets that have been emptied directly by the application.

```

public Collection<E> get(T tag) {
    Collection<E> bucket = tc.get(tag);
    if (bucket.isEmpty()) {
        tc.remove(tag);
        throw new NoSuchElementException();
    }
    return bucket;
}

```

The *getElements* method takes *tag*, the target tag, and returns a locator positioned to just before the first element in the bucket associated with the given tag. For convenience when the *tag* is not in use, an iterator for an empty bucket is returned.

```

public Locator<E> getElements(T tag) {
    if (!contains(tag))
        return emptyLocator;
    else
        return tc.get(tag).iterator();
}

```

Correctness Highlights: By the correctness of *contains*, a locator for an empty bucket is only returned when no elements are associated with the given tag. The rest of the correctness follows from the correctness of the *get* method and the bucket iterator methods. ■

The *toString* method returns a string representation for the bucket mapping, in keeping with the *toString* methods of the wrapped tagged collection and the bucket type.

```
public String toString() {
    return tc.toString();
}
```

The *accept* method takes *v*, a visitor. It traverses the entire collection on behalf of a visitor. It throws a *VisitAbortedException* when the traversal is aborted due to an exception raised by the visitor, in which case the cause held by the *VisitAbortedException* is the exception thrown by the visitor.

```
public void accept(Visitor<TaggedElement<T, Collection<E>>> v) {
    tc.accept(v);
}
```

Some applications, may need to know how many elements are mapped to a given key without iterating through the list returned by *get*. To provide such functionality, we introduce the *bucketSize* method that takes *tag*, the target tag. It returns the number of elements associated with *tag* (which by definition is 0 when the tag is not in use).

```
public int bucketSize(T tag){
    if (!tc.contains(tag))
        return 0;
    return tc.get(tag).getSize();
}
```

Representation Mutators

We now describe the representation mutators. The *ensureCapacity* and *trimToSize* methods delegate the computation to the wrapped class. Their correctness follows from that of the corresponding method call to the wrapped tagged collection.

```
public void ensureCapacity(int capacity) {tc.ensureCapacity(capacity);}
public void trimToSize() {tc.trimToSize();}
```

Content Mutators

The *put* method takes *tag*, the tag for the new element, *data*, the associated data. This method adds *data* to the bucket for the given *tag*. If no bucket exists for the given *tag*, a bucket is created using the provided factory.

```
public void put(T tag, E data) {
    if (!tc.contains(tag))
        tc.put(tag, factory.createBucket());
    tc.get(tag).add(data);
}
```

Correctness Highlights: When *tag* is not currently in use (i.e., *tc.contains(tag)* returns *false*), a new bucket is created that is associated with *tag*. In all cases, the given element is added to the bucket associated with *tag*. ■

The *putTracked* method takes *tag*, the tag for the new element, and *data*, the associated data. This method associates *data* with the given *tag* by adding it to the bucket (which is created if it does not exist) for *tag*. It returns a tracker in the bucket for the newly added element. This method throws an *UnsupportedOperationException* when the bucket type does not implement the *Tracked* interface.

```
public Locator<E> putTracked(T tag, E data) {
    if (!tc.contains(tag))
        tc.put(tag, factory.createBucket());
    Collection<E> coll = tc.get(tag);
    if (coll instanceof Tracked)
        return ((Tracked<E>) tc.get(tag)).addTracked(data);
    else
        throw new UnsupportedOperationException(
            "the bucket implementation is not tracked");
}
```

Correctness Highlights: Like that of *put* except that a locator for the new element is returned. ■

The *putAll* method takes *tc*, the tagged collection to be added to this collection.

```
public void putAll(TaggedCollection<T, E> tc) {
    for (TaggedElement<T, E> e : tc)
        put(e.getTag(), e.getElement());
}
```

The *remove* method that takes *tag*, the target tag, removes the first element in the bucket for that tag, according to the iteration order, and returns the element removed. The method throws a *NoSuchElementException* when there are no elements associated with the given *tag*.

```
public E remove(T tag) {
    Collection<E> bucket = tc.get(tag);
    if (bucket.getSize() ≤ 1)
        tc.remove(tag);
    if (bucket.getSize() > 0) {
        Iterator<E> it = bucket.iterator();
        E result = it.next();
        it.remove();
        return result;
    } else
        throw new NoSuchElementException();
}
```

The *remove* method that takes *tag*, the target tag, and *element*, the associated element to remove, removes the specified element from the bucket associated with the given tag (if it exists in that bucket). It returns *true* if and only if the element was found in the specified bucket and removed.

```

public boolean remove(T tag, E element) {
    if (tc.contains(tag)) {
        Collection<E> bucket = tc.get(tag);
        if (bucket.contains(element)) {
            bucket.remove(element);
            if (bucket.isEmpty())
                tc.remove(tag);
            return true;
        }
    }
    return false;
}

```

Correctness Highlights: The bucket class preserves NUMELEMENTS. The rest of the correctness follows directly from that of *contains* method, the *get* method, and the Bucket *contains* and *remove* methods. ■

The *removeBucket* method takes *tag*, the target tag. It removes all elements from this tagged bucket collection that are associated with the given tag. It returns the bucket that had been associated with the tag. The method throws a *NoSuchElementException* when there is no bucket for the given tag.

```

public Collection<E> removeBucket(T tag) {
    return tc.remove(tag);
}

```

The *clear* method removes all elements from this collection.

```

public void clear() {
    tc.clear();
}

```

The *iterator* method creates a new iterator that is at FORE.

```

public Iterator<TaggedElement<T, Collection<E>>> iterator() {
    return tc.iterator();
}

```

Part III

GRAPH DATA STRUCTURES AND ALGORITHMS

Chapter 51

Part III Organization

[Table 51.1](#) provides a visual summary for the organization of our coverage of the graph ADTs, data structures, and algorithms. Chapter 52 presents the Graph ADT. It begins with a case study (Section 52.1) and terminology used throughout Part III (Section 52.2). Section 52.3 presents the *Edge* interface, Section 52.4 presents the *Graph* interface, and Section 52.5 presents the *GraphRepresentation* interface. Chapter 52 closes with a discussion on selecting a graph data structure (Section 52.6) and a summary of the graph data structures (Section 52.7).

There are many efficient algorithms to perform useful computations on graphs. The Abstract-Graph class presented in Chapter 53 provides implementations for many classic graph algorithms for unweighted graphs in terms of the *Graph* interface. The advantage of this approach is that it separates the description and implementation of the algorithm from the particular implementation of the graph. Chapter 53 begins with the graph representation properties (Section 53.1), and the graph methods for modifying the graph, for iterating over the vertices and edges, and for extracting information about the graph (Section 53.2). The graph algorithms we present include algorithms to compute shortest paths in both unweighted and weighted graphs. To provide a mechanism to represent the shortest path tree computed by these algorithms, Section 53.3 presents the in-tree data structure that is returned by all graph algorithms that compute a shortest path from a specified source to all other vertices in the graph. Section 53.4 presents the breadth-first search algorithm that computes shortest paths in an unweighted graph. In Section 53.5, we present the depth-first search algorithm that can be used both to find a cycle and the connected components of a graph. Then in Section 53.6 we present precedence graphs and the topological sort algorithm. Section 53.7 presents an algorithm that uses depth-first search to compute the strongly connected components of an unweighted directed graph. Section 53.8 overviews the time complexity of the graph algorithms presented in the earlier sections. We close the chapter with a case study on garbage collection algorithms (Section 53.9) followed by a summary of related work and additional references (Section 53.10).

After presenting the graph algorithms, we present the two standard representations for an unweighted graph: the adjacency matrix data structure (Chapter 54) and the adjacency list data structure (Chapter 55).

Chapter 56 presents the weighted graph ADT. It opens with two case studies of applications that are best modeled as a weighted graph (Sections 56.1 and 56.2), and terminology that is specific to weighted graphs (Section 56.3). Section 56.4 presents the *WeightedEdge* interface, Section 56.5 presents the *SimpleWeightedEdge* class as a sample implementation for the *WeightedEdge* interface, and Section 56.6 presents the *WeightedGraph* interface. Section 56.7 discusses the trade-offs between the weighted adjacency matrix and the weighted adjacency list data structures. We present the weighted adjacency matrix in Section 56.8, and the weighted adjacency list in Section 56.9.

Following the same methodology as for unweighted graphs, Chapter 57 presents a variety of weighted graph algorithms in terms of the *WeightedGraph* interface so that any graph representation can be used easily by each algorithm. Section 57.1 presents a greedy algorithm to construct a tree. This algorithm is then instantiated to obtain both Dijkstra's shortest path algorithm (Section 57.2) and Prim's minimum spanning tree algorithm (Section 57.3). We also present Kruskal's

minimum spanning tree algorithm (Section 57.4), Bellman-Ford's single-source shortest path algorithm (Section 57.5), Floyd-Warshall's all-pairs shortest path algorithm (Section 57.7), and the Edmonds-Karp maximum flow algorithm (Section 57.8). We close with a summary of related work and additional references (Section 57.9).

Graphs	Relationships are boolean (unweighted)	Graph ADT	Chapter 52
		<i>Edge Interface</i>	Section 52.3
		<i>Graph Interface</i>	Section 52.4
		<i>GraphRepresentation Interface</i>	Section 52.5
		AbstractGraph Class	Chapter 53
		InTree Representation	Section 53.3
		Breadth First Search Algorithm	Section 53.4
	Relationships have associated weights	Depth First Search Algorithm	Section 53.5
		Topological Sort Algorithm	Section 53.6
		Strongly Connected Components Algorithm	Section 53.7
		AdjacencyMatrix Data Structure	Chapter 54
		AdjacencyList Data Structure	Chapter 55
		WeightedGraph ADT	Chapter 56
		<i>WeightedEdge Interface</i>	Section 56.4
		SimpleWeightedEdge Class	Section 56.5
		<i>WeightedGraph Interface</i>	Section 56.6
		WeightedAdjacencyMatrix Data Structure	Section 56.8
		WeightedAdjacencyList Data Structure	Section 56.9
		AbstractWeightedGraph Class	Chapter 57
		Greedy Tree Builder	Section 57.1
		Dijkstra's Single Source Shortest Path Algorithm	Section 57.2
		Prim's Minimum Spanning Tree Algorithm	Section 57.3
		Kruskal's Minimum Spanning Tree Algorithm	Section 57.4
		Bellman Ford's Single Source Shortest Path Algorithm	Section 57.5
		ShortestPathMatrix Data Structure	Section 57.6
		Floyd Warshall's All Pairs Shortest Path Algorithm	Section 57.7
		Edmonds Karp Maximum Flow Algorithm	Section 57.8

Table 51.1 An overview of the ADTs, data structures, and algorithms presented in Part III of this book.

Chapter 52

Graph ADT

A graph represents general relationships between pairs of elements from among a set of elements. The elements are called **vertices**, and the relationships between them are called **edges**. Many real-world algorithmic problems are best formulated as graph problems. Graphs are classified based on three orthogonal features.

Unweighted or Weighted Edges: Some relationships between vertices are boolean. For example, in the case study discussed in the next section, the vertices correspond to tasks, and an edge from task u to task v indicates that task u must complete before task v can begin. Such a relationship is a boolean relationship – it occurs or does not. A graph in which the relationships are boolean is called an **unweighted graph**.

Other relationships have an associated value. For example, in the graph from Section 2.1 that captures the information about exchange rates, the exchange rate from currency u to currency v is the value associated with the edge from vertex u to vertex v . A graph in which the relationships have an associated real value (or weight) is called a **weighted graph**.

While the representations for unweighted and weighted graphs are very similar, the problems for which they apply, and the corresponding algorithms are fundamentally different. Thus, we introduce the Graph ADT for unweighted graphs, and the WeightedGraph ADT for weighted graphs.

Directed or Undirected Edges: Some relationships between vertices are directional (asymmetric). A **directed graph** is used when the relationship is asymmetric. The direction of each edge indicates how the relationship is applied. Both examples mentioned above are modeled by a directed graph. For example, saying that task u must precede task v is very different from saying that task v must precede task u . So an edge from u to v means something different than an edge from v to u . When relationships are symmetric (i.e., the two elements can be interchanged arbitrarily), an **undirected graph** is used. For example, consider a graph with a vertex for each park within a city, and where an edge between vertices u and v represents the existence of a bike path between park u and park v . Assuming all bike paths permit two-way traffic, the graph is undirected. Saying there is a bike path between park u and park v is equivalent to saying there is a bike path between park v and park u .

Graph or Multigraph: Another graph feature concerns whether or not there can be more than one edge between vertices u and v in an undirected graph, or from vertex u to vertex v in a directed graph. For example, in the graph to represent the exchange rates between currencies, there is a single exchange rate between currency u and currency v , so there would be only a single edge from u to v , and a single edge from v to u .

In a **multigraph** there can be more than one edge between the same pair of vertices in an undirected graph, or multiple edges from vertex u to vertex v in a directed graph. Section 56.1 presents a case study in which a graph is used to model commercial flight information for a set of airports. Here, a vertex represents any airport and an edge from u to v represents a

flight from airport u to airport v . Since there can be many different flights (with different costs, departure times, etc.) from one airport to another, such an example would be modeled best by a directed weighted multigraph.

We use the following naming conventions.

Graph: Unweighted graphs are further classified as follows.

Undirected Graph - Unweighted graph with undirected edges, for which there can be at most one edge between vertex u and vertex v .

Directed Graph - Unweighted graph with directed edges, for which there can be at most one edge from vertex u to vertex v .

Multigraph - Unweighted graph with undirected edges, for which there can be any number of edges between vertex u and vertex v . While it is somewhat uncommon to have an unweighted multigraph, as an example, suppose there are several distinct bike paths between park u and park v . If a bike path is closed for resurfacing, then it could be useful to have a representation in which each distinct bike path between two parks is represented by a different edge.

Directed Multigraph - Unweighted graph with directed edges, for which there can be any number of edges from vertex u to vertex v .

Weighted Graph: Weighted graphs are further classified as follows.

Undirected Weighted Graph - Weighted graph with undirected edges, for which there can be at most one edge between vertex u and vertex v .

Weighted Directed Graph - Weighted graph with directed edges, for which there can be at most one edge from vertex u to vertex v .

Weighted Multigraph - Weighted graph with undirected edges, for which there can be any number of edges between vertex u and vertex v .

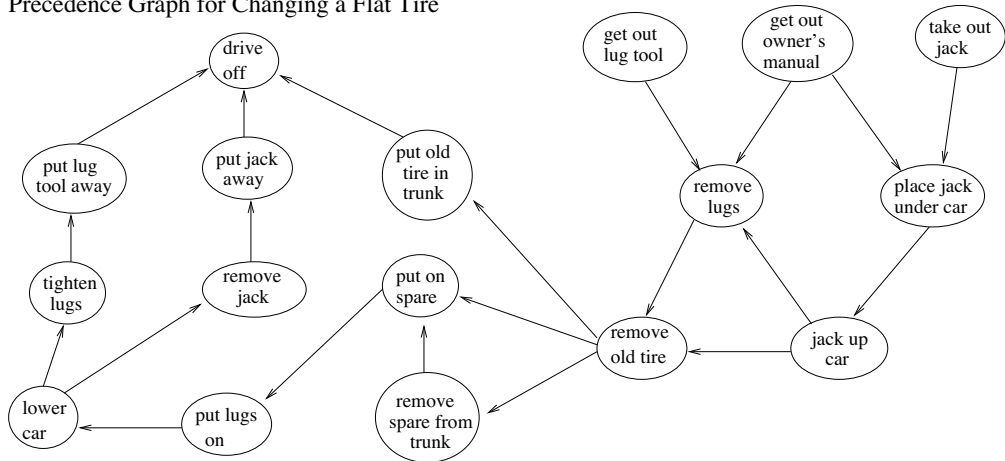
Weighted Directed Multigraph - Weighted graph with directed edges, for which there can be any number of edges from vertex u to vertex v .

52.1 Case Study: Task Scheduler

Consider a set of tasks $t_1, \dots, t_n$ that all must be completed. Suppose there are constraints on the order in which the tasks can be completed. In particular, there are precedence constraints of the form that task t_i must complete before task t_j can begin. If we define a graph whose vertices are the tasks, each such precedence constraint can be represented by an unweighted directed edge from t_i to t_j . Such a graph is generally called a **precedence graph**. Since there would be no reason to include the same constraint twice, the resulting graph is a directed unweighted graph. Note that if there is a directed cycle in the graph, then there is no way to complete all of the tasks while obeying all of the precedence constraints. Thus, a precedence graph should be acyclic. [Figure 52.1](#) shows a precedence graph for the tasks involved in changing a flat tire.

A typical problem faced by applications involving tasks and precedence constraints is to find some order in which to perform the tasks that does not violate any precedence constraint. Such an ordering of the tasks is called a **topological order** since the topology of the graph implies that if the vertices are placed in a line, left to right, consistent with this order, then all of the edges are directed from the left to the right. In Section 53.5, we present an algorithm to solve this problem.

Precedence Graph for Changing a Flat Tire

**Figure 52.1**

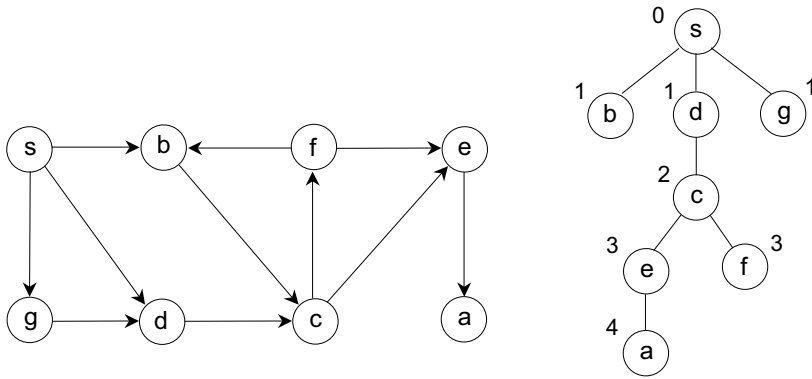
A sample precedence graph.

52.2 Terminology

Throughout this chapter, we use the following terminology.

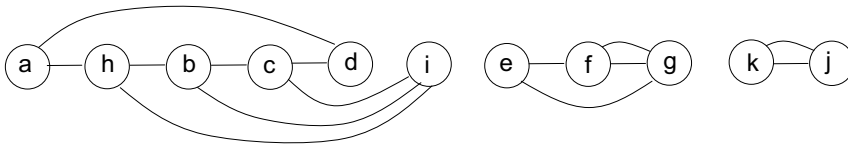
- A graph $G = (V, E)$ where V is the set of vertices and E is the set of edges. We use $n = |V|$ to denote the number of vertices, and $m = |E|$ to denote the number of edges.
- In an undirected graph, a **multi-edge** between u and v is a set of two or more undirected edges between u and v .
- In a directed graph, a **multi-edge** from u to v is set of two or more directed edges from u to v .
- A **multigraph** is a graph that has at least one pair of vertices with a multi-edge.
- An undirected edge between u and v is denoted as the **unordered pair** $\{u, v\}$.
- A directed edge from u to v is denoted as the **ordered pair** (u, v) where u is called the **source**, and v is called the **destination**. Another common terminology is to call u the **tail** and v the **head**.
- We say that an edge e is **incident** to vertex v if $e.source = v$ or $e.dest = v$. Alternatively, the term **adjacent** is sometimes used.
- A **path** in an undirected graph $G = (V, E)$ is a sequence of undirected edges $P = \langle e_1, e_2, \dots, e_k \rangle$, possibly containing duplicates, where the orientation for $e_i \in P$ is specified such that $e_i.dest = e_{i+1}.source$. We say that the path is between vertex $e_1.source$ and vertex $e_k.dest$. For a graph that is not a multigraph, a path can also be specified by the sequence of vertices

$$P = \langle e_1.source, e_2.source, \dots, e_k.source, e_k.dest \rangle.$$

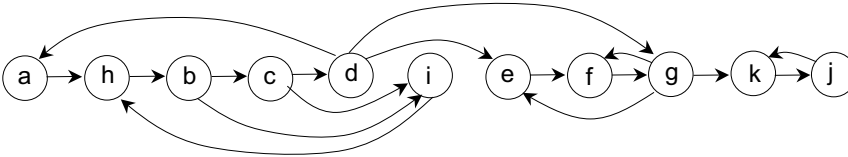
**Figure 52.2**

A sample directed graph and the shortest path tree for source vertex s . Next to each vertex in the shortest path tree is the length of the shortest path from s to that vertex.

- A **path** in a directed graph $G = (V, E)$ is a sequence of directed edges $P = \langle e_1, e_2, \dots, e_k \rangle$, possibly containing duplicates, where for each $e_i \in P$, $e_i.dest = e_{i+1.source}$. We say that the path is from vertex $e_1.source$ to vertex $e_k.dest$. For a directed graph that is not a multigraph, a path can also be specified by the sequence of vertices on the path.
- Vertex v **occurs** in P , if there is some edge e_i in P incident to v .
- We define the **length** of path P as the number of edges in the path, and denote it by $|P|$.
- A **shortest unweighted path** from s to v is a path P from s to v such that for any other path P' from s to v , $|P| \leq |P'|$.
- A **shortest path tree** T is an in-tree (see page 13) that is used to represent the shortest path from a specified source vertex s to all other vertices in the graph that are reachable from s . The path from s to any vertex v in T (obtained by tracing the path in reverse order from v to the root s) is a shortest path from s to v . Figure 52.2 shows a shortest path tree for a directed graph.
- For vertex v in shortest path tree T , we use $v.edgeFromParent$ to denote the edge in the graph from v 's parent to v . We say that the path **defined** by T from s to v is the path defined by repeatedly following $edgeFromParent$, and inserting that edge to the front of the path until s is reached. For example, in the shortest path tree shown in Figure 52.2, for f , $f.edgeFromParent$ would be the edge (c, f) , and for c , $c.edgeFromParent$ would be the edge (d, c) , and for d , $d.edgeFromParent$ would be the edge (s, d) . Thus the path defined from s to f is $s \rightarrow d \rightarrow c \rightarrow f$, which has length 3.
- A path is a **simple path** if no vertex occurs more than once.
- A **cycle** is a path C that starts and ends at the same vertex v . When no vertex occurs more than once in C (except that v occurs once at the start and once at the end), it is called a **simple cycle**.
- We say that a graph is **cyclic** if it has at least once cycle, and **acyclic** if it has no cycles.
- We say that u is **reachable** from v if there is a path from u to v in the graph.

**Figure 52.3**

This undirected graph has connected components $\{\{a, h, b, c, d, i\}, \{e, f, g\}, \{k, j\}\}$.

**Figure 52.4**

This directed graph has strongly connected components $\{\{a, h, b, c, d, i\}, \{e, f, g\}, \{k, j\}\}$.

- We define the **reachability relation**

$$R = \{(u, v) \mid u \text{ is reachable from } v \text{ and } v \text{ is reachable from } u\}.$$

- A **connected component** in an undirected graph is a maximal set of vertices that are mutually reachable from each other. In other words, the connected components are the equivalence classes of the relation R . The undirected graph shown in Figure 52.3 has three connected components.
- We say that an undirected graph is **connected** when it has a single connected component.
- A **strongly connected component** in a directed graph is defined as a maximal set vertices that are mutually reachable from each other. The directed graph shown in Figure 52.3 has three strongly connected components. In an acyclic directed graph, each vertex forms its own strongly connected component.
- A **self loop** at vertex u is an edge from u to itself.
- The **degree** of vertex v is the number of edges incident to v . For example, vertex h has degree 3 in both the undirected graph of Figure 52.3 and the directed graph of Figure 52.4. A self loop at vertex v contributes two to the degree of v .
- For a directed graph $G = (V, E)$, we define the **outedges** of vertex v as $\{e \in E \mid e.source = v\}$. Likewise, we define the **inedges** of vertex v to be $\{e \in E \mid e.dest = v\}$. For example, in the directed graph of Figure 52.4, vertex g has outedges (g, e) , (g, f) , and (g, k) , and inedges (d, g) and (f, g) .
- For an undirected graph, we define both the **outedges** and the **inedges** of v to be all edges that are incident to v .
- For a directed graph, the **indegree** of vertex v is defined as its number of inedges, and the **outdegree** of vertex v is its number of outedges. We let $d_{out}(v)$ and $d_{in}(v)$ denote the indegree and out degree of v , respectively. A self loop (v, v) contributes one to the indegree of v , and one to the outdegree of v . Observe that the degree of vertex v in a directed graph is the sum of its indegree and outdegree. As an example, in the graph of Figure 52.4, vertex g has outdegree 3, indegree 2, and degree 5.

- We say that a graph is **dense** if the number of edges in the graph is at least $n^2/4$. While the specific cutoff is somewhat arbitrary, the intuition is that a significant fraction of the potential edges in the graph exist.
- We say that a graph is **sparse** if the number of edges in the graph is asymptotically less than n^2 . For example, if each vertex, on average, had only a constant number of edges, then the graph would be a sparse graph. In contrast, in a dense graph, the average degree for a vertex is $\Omega(n)$.

52.3 Edge Interface

Edge<V>

All graph representations maintain a set of edges, with the difference being the way in which the edges are organized. The primary change to represent a weighted graph will be to extend *Edge* to *WeightedEdge* that also associates a real-valued weight with each edge. The source and destination of an edge are required to be immutable.

V dest(): Returns the destination (head) of this edge.

V source(): Returns the source (tail) of this edge.

The SimpleEdge class provides a sample implementation for an edge.

```
public class SimpleEdge<V> implements Edge<V> {
    V head;
    V tail;

    public SimpleEdge(V head, V tail) {
        this.head = head;
        this.tail = tail;
    }
    public V source() {return head;}
    public V dest() {return tail;}
}
```

52.4 Graph Interface

The graph algorithms we present do not depend on a particular graph implementation. Instead, they are implemented entirely in terms of either the Graph ADT or WeightedGraph ADT (Chapter 56). One advantage of this approach, which uses the **bridge design pattern**, is that it allows the presentation and implementation of the graph algorithms to be expressed independent of the underlying graph representation. In this section, we describe the *Graph* interface that provides the methods required to create and modify a graph, and methods to perform commonly needed computation on

the graph, such as computing a shortest path tree for a specified source, determining if a graph is cyclic, or finding a valid topological order for a precedence graph.

The following list includes the constructor and *Graph* interface methods for modifying the graph, for iterating over the vertices and edges, and for extracting information about the graph. In addition, we define a *GraphException* (that simply extends *Exception*) for any exception that is specific to graphs.

Graph(GraphRepresentation<V,E> rep, boolean directed, boolean multigraph): Creates a new empty graph that uses the given graph representation. The second and third parameters specify whether or not the graph is directed, and whether or not it is a multigraph. Two possible graph representations, the adjacency matrix and the adjacency list, are discussed in Section 52.6.

void addEdge(E edge): Adds the given edge to this graph. If the graph does not allow multi-edges, this method requires that there is not already an edge from the source to the destination.

boolean addVertex(V vertex): Adds the given vertex to this graph. It returns *true* if the vertex is successfully added, and *false* if the vertex already existed in the graph.

boolean allowsMultiEdges(): Returns *true* if and only if this graph allows multi-edges.

boolean containsEdge(V source, V dest): Returns *true* if and only if there is an edge from *source* to *dest* in this graph.

boolean containsVertex(V vertex): Returns *true* if and only if the given vertex is in this graph.

Iterator<E> edgesFrom(V vertex): Returns an iterator over the outgoing edges from the given vertex.

Iterator<E> edgesTo(V vertex): Returns an iterator over the incoming edges for the given vertex. This is an optional operation for directed graphs.

E getEdge(V source, V dest): Returns an edge in the graph from *source* to *dest*, or *null* if there is no such edge.

boolean isDirected(): Returns *true* if and only if this graph is a directed graph.

NonMutatingIterator<V> iterator(): Returns a non-mutating iterator over the vertices in this graph.

int numVertices(): Returns the number of vertices in this graph.

boolean removeEdge(E edge): Removes the given edge from this graph. It returns *true* when the edge is successfully removed, and *false* if the edge did not exist in the graph.

boolean removeVertex(V vertex): Removes the given vertex from this graph. It returns *true* when the vertex is successfully removed, and *false* if the vertex did not exist in the graph. Removing a vertex implicitly removes all of its incident edges. Because edges may be deleted lazily in some implementations on the basis of whether or not the incident vertices are still in the graph, this method requires that the removed vertex (or any equivalent one) is never added back into the graph.

The following methods perform computation on the graph.

Set<Set<V>> getConnectedComponents(): Returns a set holding the set of vertices in each connected component of this graph. This method is applicable only for undirected graphs. It throws a *GraphException* when the graph is directed since the connected components are defined only for an undirected graph.

PositionalCollection<E> getCycle(): Returns a positional collection containing the edges in some cycle in this graph, in the order they appear in the cycle. It throws a *GraphException* if the graph is acyclic. The *hasCycle* method can be used to determine if the graph has a cycle.

Set<Set<V>> getStronglyConnectedComponents(): Returns a set holding the set of vertices in each strongly connected component of this graph. This method is applicable only for directed graphs. It throws a *GraphException* when the graph is undirected since the strongly connected components are defined only for a directed graph.

boolean hasCycle(): Returns *true* if and only this graph has a cycle.

int numConnectedComponents(): Returns the number of connected components in this graph. It throws a *GraphException* when the graph is directed since the connected components are defined only for an undirected graph.

int numStronglyConnectedComponents(): Returns the number of strongly connected components in this graph. It throws a *GraphException* when the graph is undirected since the strongly connected components are defined only for a directed graph.

PositionalCollection<V> topologicalOrder(): Returns a positional collection that holds a permutation of the vertices of a directed graph in a valid topological order. It throws a *GraphException* if the graph is undirected, or if there is a directed cycle.

InTree<V,E> unweightedShortestPaths(V source): Uses breadth-first search to compute and return a shortest path tree for the given source vertex.

52.5 Graph Representation Interface

package graph

Iterable<V>

↳ **GraphRepresentation<V, E extend Edge<V>>**

The *GraphRepresentation* interface presented in this chapter defines the methods that must be supported by any graph representation, such as the adjacency list and adjacency matrix representations. Our approach illustrates the bridge design pattern, since different types of graphs (e.g., weighted and unweighted) can independently choose their internal representation from among the classes that implement *GraphRepresentation*.

void addEdge(E edge): Adds the given edge to this graph.

boolean addVertex(V vertex): Adds the given vertex to this graph unless it is already in the graph. It returns *true* if and only if the vertex is added.

boolean containsEdge(V source, V dest): Returns *true* if and only if there is an edge in the graph from the source vertex to the destination vertex.

boolean containsVertex(V vertex): Returns *true* if and only if the give vertex is in the graph.

NonMutatingIterator<E> edgesFrom(V source): Returns an iterator over the outgoing edges from *source*. Observe that for an undirected graph, the iterator is over all edges incident to *source*.

NonMutatingIterator<E> edgesTo(V dest): Returns an iterator over the incoming edges from *dest*. This method throws an *UnsupportedOperationException* when the graph representations does not support this capability.

E getEdge(V source, V dest): Returns an edge in the graph from *source* to *dest*, or *null* if there is no such edge.

boolean isDirected(): Returns *true* if and only if this graph representation is being used to support a directed graph.

NonMutatingIterator<V> iterator(): Returns a non-mutating iterator over the vertices of the graph. A non-mutating iterator is returned to ensure that the application program can only remove a vertex from a graph using the *removeVertex* method.

int numVertices(): Returns the number of vertices in this graph.

boolean removeEdge(E edge): Returns *true* when the given edge is successfully removed, and *false*, if the edge did not exist.

boolean removeVertex(V vertex): Returns *true* if and only if the given vertex existed and was removed.

52.6 Selecting a Data Structure

The simplest representation for a graph is an **edge set**, which is simply a set containing all the edges. A very common operation used within graph algorithms is to iterate through all edges with a given vertex as the source vertex. A drawback of an edge set is that there is no efficient support for such an operation, so a more structured approach is preferable.

There are two common graph representations: the adjacency matrix and the adjacency list. We first describe these two data structures for a directed unweighted graph, and then discuss how they can be generalized for other types of graphs. Throughout this section, we assume for simplicity that there are n vertices $v_0, \dots, v_{n-1}$ in the graph*.

An **adjacency matrix** is a two-dimensional $n \times n$ array M where the entry in row i and column j corresponds to directed edge (v_i, v_j) . In its simplest form, $M[i][j]$ is a boolean that is *true* if and only if the edge (v_i, v_j) exists in the graph. Typically, however there is more information associated with each edge than the source and destination vertex. In Section 52.3 we introduced the *Edge* interface, which any class can implement to define the type for each edge in a graph. Therefore, any object (airline flight, bike path, pipeline, exchange rate, etc.) could be used as an edge in a graph.

*In Section 52.5, we discuss these data structures in much more depth and provide implementations. The implementation for the adjacency matrix includes a mechanism to associate an integer id with each vertex.

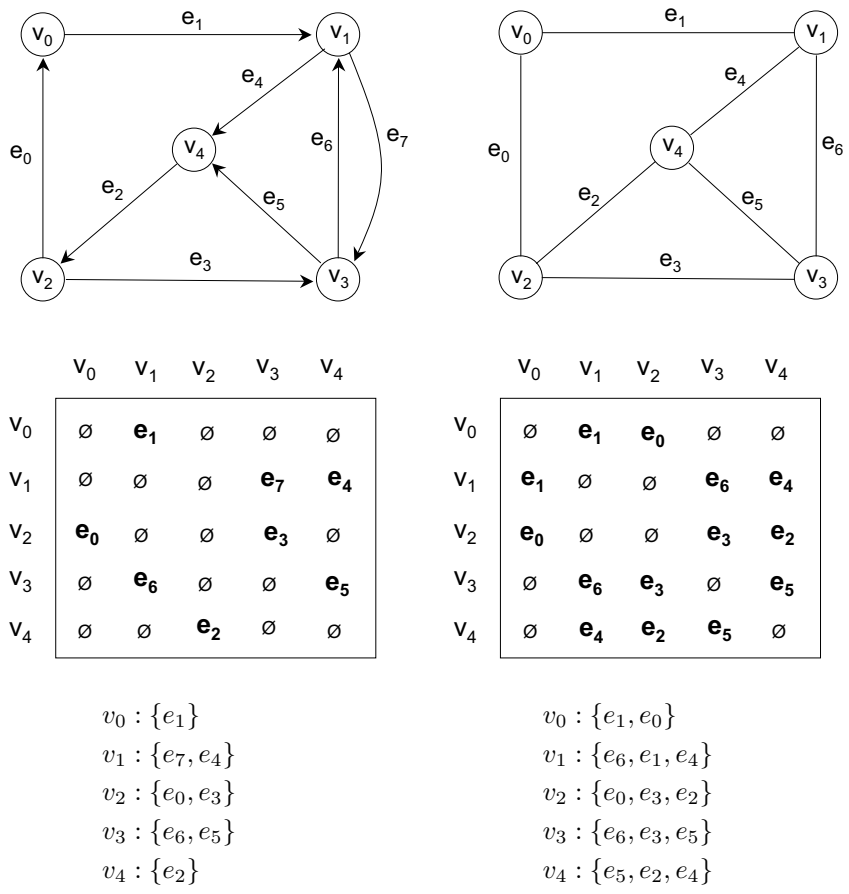


Figure 52.5
Top Row: a directed graph (left) and undirected graph (right). Middle Row: visualization of the adjacency matrix representation. Bottom Row: visualization of the adjacency list representation.

In our adjacency matrix implementation $M[i][j]$ is either *null* if there is not an edge from v_i to v_j , or otherwise it is a reference to the edge instance corresponding to the edge from v_i to v_j .

An **adjacency list** is a mapping from vertices to lists (or more generally sets). More specifically, vertex v_i is mapped to the set of its outedges. If there are no edges in the graph with v_i as a source, then v_i is not included in the mapping.

To select between an adjacency matrix and an adjacency list, consider which of the following properties are most important for the application.

Ability to efficiently iterate through the outedges of v_i . Since an adjacency list holds a list of the outedges for each vertex, the time complexity to iterate through the outedges of v_i is $O(d_{\text{out}}(v_i) + 1)$. In contrast, for an adjacency matrix, all n entries in row i must be examined, leading to an $O(n)$ time complexity, regardless of $d_{\text{out}}(v_i)$. So when $d_{\text{out}}(v_i) = 1$, this operation takes constant time for an adjacency list and linear time for an adjacency matrix.

Ability to efficiently iterate through the inedges of v_i . For an adjacency matrix, all n entries in column i must be examined, leading to an $O(n)$ time complexity. Since an adjacency list is organized based on the outedges of each vertex, it provides no direct support for iterating over the inedges of a vertex. In particular, it would be necessary to iterate over all vertices of the

graph, and for each vertex examine each outgoing edge to determine if its destination is v_i , which takes $O(m + n)$ time overall. However, if the other advantages of an adjacency list are important, then this drawback can be addressed at the cost of doubling the space usage. To this end, we define an **augmented adjacency list** as a variation of an adjacency list that stores both a list of outedges and a list of inedges for each vertex. For an augmented adjacency list, the time complexity for iterating over all inedges of vertex v_i is $O(d_{\text{in}}(v_i) + 1)$.

Ability to efficiently determine if (v_i, v_j) is in the graph. An adjacency matrix is very well-suited for this operation since $M[i][j]$ is accessible in constant time and directly references such an edge if it exists. For an adjacency list, checking for the existence of an edge (v_i, v_j) takes $O(d_{\text{out}}(v_i) + 1)$ time. For an augmented adjacency list, there is only a minimal improvement to $O(\min(d_{\text{out}}(v_i), d_{\text{in}}(v_j)) + 1)$ since the shorter of the two lists can be traversed.

Space usage. An adjacency matrix requires n^2 references. An adjacency list uses $4m + n$ references, namely the n head pointers and a list item for each of the m edges, where each list item has a reference to the edge, a next reference, a previous reference, and a reference to its type. Thus, when $m \leq (n^2 - n)/4$, an adjacency list is more space efficient.

Removal of an edge. The time to remove an edge $e = (v_i, v_j)$ is dominated by the time to locate the edge. As discussed above, this operation takes $O(1)$ time in an adjacency matrix, and $O(d_{\text{out}}(v_i) + 1)$ for an adjacency list. For an augmented adjacency list, the edge must be removed in both places, so the cost to remove an edge is $O(d_{\text{out}}(v_i) + d_{\text{in}}(v_j) + 1)$. To save the cost of finding the edge a second time, each edge could maintain a tracker to its “mate,” or a doubly threaded list technique could be used, as discussed on page 904. Either of these techniques would reduce the cost of removing an edge from an augmented adjacency list to $O(\min(d_{\text{out}}(v_i), d_{\text{in}}(v_j)) + 1)$.

Removal of a vertex. Removing a vertex is more complex for both representations. For an adjacency matrix, it would take $O(n^2)$ time to reallocate a smaller matrix, and copy the entries. We have instead chosen to implement a **lazy delete** in which the vertex is just marked as deleted, and the corresponding row and column have all entries set to *null*, but remain in the matrix. If a new vertex is later added to the graph, then this space in the adjacency matrix is reused for the new vertex. However, if there is a significant decrease in the number of vertices in the graph, then the matrix would need to be resized.

For an adjacency list, the affect on the space usage is not as large. In our implementation, the list of outedges is kept as a bucket mapping (Chapter 50), making it possible to remove the list of outedges for the deleted vertex from the mapping. However, the edge objects that have the removed vertex as a destination cannot be efficiently located. In our implementation we lazily delete these list elements. Whenever iterating over an edge list, any edges incident to deleted vertices are removed so they do not continue to waste space.

So far we have discussed only how an adjacency matrix and adjacency list compare when used to represent a directed unweighted graph. We briefly discuss the changes for other forms of graphs.

Undirected Graph: For an adjacency list, an undirected graph of m edges is represented as a directed graph of at most $2m$ edges by representing undirected edge $\{u, v\}$ (for $u \neq v$) by the two directed edges (u, v) and (v, u) . A self loop $\{u, u\}$ is just represented by the single directed edge (u, u) .

For an adjacency matrix, one possibility is to allocate row i of the matrix to have just i entries, which would cut the space usage roughly in half. However, such an approach would make iteration slightly more difficult. For simplicity, we have chosen to place each undirected edge $\{v_i, v_j\}$ in both $M[i][j]$ and $M[j][i]$.

Multiple Edges: The adjacency list easily accommodates multiple edges, by just adding them to the list of outedges. For the adjacency matrix, multiple edges are handled by letting each entry in the matrix refer to a list of edges between the corresponding vertices. To minimize the space usage, each matrix entry $M[i][j]$ is converted to a list only if there are at least two edges from v_i to v_j .

Weighted Edges: For both the adjacency matrix and adjacency list, the only change needed for weighted edges is to modify the edge class to include a weight. The *WeightedEdge* interface is presented in Section 56.4.

52.7 Summary of Graph Data Structures

This book provides the following unweighted graph implementations. This section summarizes the strengths and weaknesses of each of these in terms of application requirements. [Table 52.6](#) provides a visual summary of these trade-offs.

AdjacencyMatrix: The adjacency matrix is the most space efficient representation for dense graphs (roughly $m \geq n^2/4$). Another advantage of an adjacency matrix is that it takes constant time to determine if there is some edge from vertex v_i to vertex v_j . It takes $O(n)$ time to iterate through all outgoing edges from v , or all incoming edges into vertex v . For a sparse graph, an adjacency matrix uses more space than an adjacency list, and also it takes asymptotically longer to iterate through the outgoing edges among all vertices, which in turn can increase the time complexity of many graph algorithms.

AdjacencyList: The adjacency list is the most space efficient representation for a sparse graph. Another important advantage of an adjacency list is that it takes $O(d_{\text{out}}(v) + 1)$ time to iterate through all outgoing edges of vertex v . Many graph algorithms iterate through the outgoing edges of all the vertices. This takes $O(n + m)$ time for an adjacency list, versus $O(n^2)$ time for an adjacency matrix. Another advantage of an adjacency list is that its iteration order is consistent with the order in which the edges were added. The adjacency matrix and adjacency set (below), do not provide such a guarantee on the iteration order. There are two significant drawbacks of an adjacency list. First, it takes $O(n + m)$ time to iterate through all incoming edges from vertex v . Second, it takes $O(d_{\text{out}}(v_i) + 1)$ to determine if there is an edge from vertex v_i to vertex v_j .

AugmentedAdjacencyList: An augmented adjacency list, which augments an adjacency list to maintain a list of incoming edges for each vertex, uses twice as much space as standard adjacency list. However, for a graph with fewer than about $n^2/8$ edges, it is still more space efficient than an adjacency matrix. Since this data structure keeps both the standard “out-edges” adjacency list and also the “in-edges” adjacency list, it takes twice as much time to add an edge or remove an edge. The advantage of an augmented adjacency list is that it supports iterating through all inedges of vertex v in $O(d_{\text{in}}(v) + 1)$ time. It also improves the cost for determining if there is an edge from vertex v_i to vertex v_j , but an adjacency matrix is still much more efficient for that operation. Another advantage of an augmented adjacency list is that the iteration order for both the outgoing and incoming edges from vertex v is consistent with the order in which they were added to the graph.

AdjacencySet: An adjacency set is just like an adjacency list, except that the list of outgoing edges is replaced by a set of outgoing edges. This change does not have any significant affect

Key						
⦿	Excellent					
○	Very Good					
⊖	Good					
●	Fair					
⊗	Unsupported method					
Method		AdjacencyMatrix	AdjacencyList	AugmentedAdjacencyList	AdjacencySet	Augmented AdjacencySet
Public Methods to Access and Modify the Graph	addEdge(e)	⦿	⦿	⦿	○	○
	addVertex(v)	⦿	⦿	⦿	⦿	⦿
	containsEdge(source, dest)	⦿	●	●	○	○
	containsVertex(v)	⦿	⦿	⦿	⦿	⦿
	edgesFrom(v), cost per edge to iterate	⊖	⦿	⦿	⦿	⦿
	edgesTo(v), cost per edge to iterate	⊖	⊗	⦿	⊗	⦿
	isDirected(), isMultiGraph(), numVertices()	⦿	⦿	⦿	⦿	⦿
	iterator(), cost per vertex to iterator	⦿	⦿	⦿	⦿	⦿
	removeEdge(e)	⦿	●	●	○	○
	removeVertex(v)	●	○	○	○	○
Public Methods to Perform Computations on the Graph	unweightedShortestPath(source), cost per edge (uses breadth-first search to compute and return a shortest path tree)	⊖	⦿	⦿	⦿	⦿
	depthFirstSearch(), cost per graph edge, for preprocessing* to support the following methods	⊖	⦿	⦿	⦿	⦿
	getConnectedComponents()					
	hasCycle(), getCycle(), per vertex in cycle	⦿	⦿	⦿	⦿	⦿
	numComponents()					
	topologicalOrder()					
Other Issues	stronglyConnectedComponents(), cost per graph edge, for preprocessing* to support the following methods	⊖	⦿	⦿	⦿	⦿
	getStronglyConnectedComponents()	⦿	⦿	⦿	⦿	⦿
	numStronglyConnectedComponents()					
	iteration order for edges guaranteed to be in insertion order					
Time Complexity		⦿	worst-case $O(1)$ time	⦿	$c(n+m)$	
		○	expected-case $O(1)$ time	○	$2c(n+m)$	
		⊖	$O(n/d)$ where d is the number of edges involved	⊖	$O(n^2)$	
		●	$O(n)$	where c is a constant		
		⊗	not supported, though could be done in $O(n+m)$ time			

*preprocessing is performed the first time a method from this group is called after a graph mutation

Table 52.6 Trade-offs among the data structures that implement the Graph ADT.

on the space usage. It slightly increases the cost of iterating over the outgoing edges, but this operation still has expected constant cost per edge, as long as the set implementation chooses an appropriate load factor. The big advantage of this modification is it takes expected constant time to determine if there is some edge from vertex v_i to vertex v_j .

AugmentedAdjacencySet: Just as an adjacency list can be augmented to maintain a list of incoming edges for each vertex, an adjacency set can be augmented to maintain a set of incoming edges for each vertex. The augmented adjacency set supports its operations very efficiently, and is more space efficient than an adjacency matrix roughly when $m < n^2/8$.

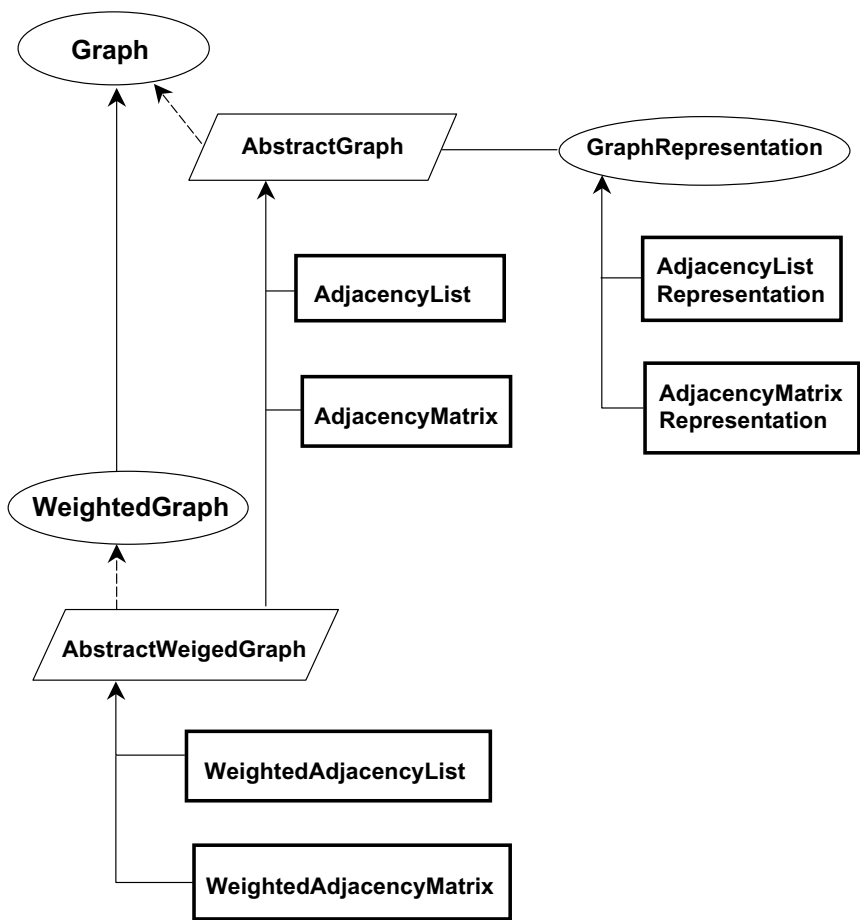


Figure 52.7
The class hierarchy for the Graph ADT and Weighted Graph ADT classes and graph representations.

Figure 52.7 shows the class hierarchy for the graph and graph representation data structures presented in this book for both the unweighted and weighted graph. The weighted graph ADT is discussed in more depth in Chapter 56.

Chapter 53

Abstract Graph and Graph Algorithms

package graph

AbstractGraph<V, E extends **Edge**<V>> implements **Graph**<V,E>

The **AbstractGraph** class embodies algorithms that perform computations on a graph in terms of the **Graph** interface. This allows subclasses that use any graph representation to benefit from these algorithms. We present an implementation for an adjacency matrix in Chapter 54, and an implementation for an adjacency list in Chapter 55. Although those representations are very different, the algorithms we present in this chapter work for both.

Uses: Our graph algorithm implementations use many of the other data structures presented in this book, including:

CircularArray (Chapter 12) is used to hold the topological order of an acyclic directed graph, and for a directed cycle in a cyclic directed graph.

Queue (Chapter 18) is used for breadth-first search.

OpenAddressing (Chapter 22) is used for the set of edges returned by *edgeSet*, and to store each individual component and the set of components for the strongly connected components algorithm.

OpenAddressingMapping (Section 49.7.2) is used for the mapping from each vertex to its associated data in the depth-first search forest.

Instance Variables and Constants: The instance variable *graph* is the graph representation that is passed to the abstract graph constructor. The graph representation can be for either a directed or an undirected graph. In addition, *multigraph* is *true* if the graph allows multiple edges between the same pair of vertices (and in the same direction for a directed graph), and *false* otherwise.

```
GraphRepresentation<V, E> graph;                                //the graph
boolean multigraph;                                             //true iff its a multigraph
```

The instance variable *components* is a set that contains one set of vertices for each connected component in the graph.

```
Set<Set<V>> components;
```

Since there are several problems that can be solved by depth first search, the result of depth-first search is stored in instance variables that need only be recomputed when the graph is updated.

```
Mapping<V, DFSData<E>> dfsForest;                             //data associated with each vertex by dfs
PositionalCollection<V> topOrder;                               //a valid topological order for an acyclic graph
PositionalCollection<E> directedCycle;                         //a directed cycle in a cyclic graph
```

Similar to the instance variables used to hold the results from *dfs*, *scc* holds the set of the strongly connected components of this graph.

```
Set<Set<V>> scc;
```

53.1 Representation Properties

Many graph algorithms, such as depth first search (Section 53.5) and strongly connected components (Section 53.7), produce data structures as the results of their computations. These data structures can be used to answer multiple queries without the need to execute the graph algorithms again. Therefore, for improved performance, the instance variables *dfsData*, *components topOrder*, *directedCycle*, and *scc* hold references to these data structures to streamline future queries. However, when the graph is modified, it is necessary to discard the results of these old computations, for they would no longer be consistent with the graph and therefore would be useless for answering queries. We introduce the *UPDATED* property to provide an indication when computations must be repeated because the graph has been changed. Also, discarding the results of old computations allows that space to be reclaimed by the garbage collector, unless the user program retains a reference.

UPDATED: If the graph has been modified since the *dfs* method has been executed then *dfsData*, *components*, *topOrder*, and *directedCycle* are *null*. Otherwise, they are not *null*. Likewise, *scc* is *null* if and only if the graph has been created or modified since the *stronglyConnectedComponents* method has been executed.

The specific invariants used within the correctness proofs for the graph algorithms are presented in the individual sections.

53.2 Methods

The constructor takes as parameters *graph*, the graph representation, and *multigraph*, *true* if the graph allows multi-edges. It creates an abstract graph with the given specification.

```
public AbstractGraph(GraphRepresentation<V, E> graph, boolean multigraph){
    this.graph = graph;
    this.multigraph = multigraph;
}
```

Before presenting the graph algorithms, we define a number of methods that support users of the graph and also simplify the presentation of the algorithms by helping to make the code more readable. The *isDirected* method returns *true* if and only if the graph is directed.

```
public boolean isDirected() {
    return graph.isDirected();
}
```

The *allowsMultiEdges* method returns *true* if and only if the graph is a multigraph.

```

public boolean allowsMultiEdges() {
    return multigraph;
}

```

The *numVertices* method returns the number of vertices in this graph.

```

public int numVertices() {
    return graph.numVertices();
}

```

The *containsVertex* method takes *v*, the target vertex. It returns *true* if and only if *v* is in this graph.

```

public boolean containsVertex(V v) {
    return graph.containsVertex(v);
}

```

The *containsEdge* method takes *source*, the source vertex, and *dest*, the destination vertex. It returns *true* if and only if there is an edge in the graph from the source vertex to the destination vertex.

```

public boolean containsEdge(V source, V dest) {
    return graph.containsEdge(source, dest);
}

```

The *getEdge* method takes *source*, the source vertex, and *dest*, the destination vertex. It returns an edge in the graph from the source vertex to the destination vertex, or *null* if no such edge exists.

```

public E getEdge(V source, V dest) {
    return graph.getEdge(source, dest);
}

```

The internal *neighbor* method takes *v*, a vertex, and *e*, one of its incident edges. It returns the other endpoint of edge *e*. This method requires that *v* is one of the two endpoints of *e*.

```

static<V, E extends Edge<V>> V neighbor(V v, E e) {
    if (e.source() == v)
        return e.dest();
    else
        return e.source();
}

```

The internal *graphUpdated* method preserves UPDATED by resetting to *null* all variables that are invalidated when the graph is modified.

```

void graphUpdated() {
    dfsForest = null;           //set by depth-first search
    topOrder = null;           //set by depth-first search
    directedCycle = null;      //set by depth-first search
    components = null;         //set by depth-first search
    scc = null;                //set by the strongly connected components algorithm
}

```

The *addVertex* method takes *v*, the vertex to add to this graph. It returns *true* if *v* is added to this graph, or *false* if *v* is already in this graph.

```

public boolean addVertex(V v) {
    if (!graph.addVertex(v))
        return false;                                //vertex is already in the graph
    graphUpdated();                                //preserve Updated
    return true;
}

```

Correctness Highlights: The call to *graphUpdated* when the graph is modified preserves UPDATED. The rest of the correctness follows from *addVertex* of the *GraphRepresentation* class. ■

The *removeVertex* method takes *v*, the vertex to remove. It returns *true* if *v* is removed, or *false* if *v* is not in the graph, and so cannot be removed. This method requires the removed vertex (or any equivalent vertex) is never added back into the graph because edges are deleted lazily in some implementations on the basis of whether or not the incident vertices are still in the graph.

```

public boolean removeVertex(V v) {
    if (!graph.removeVertex(v))
        return false;                                //vertex is not in graph to remove
    graphUpdated();                                //preserve Updated
    return true;
}

```

Correctness Highlights: The call to *graphUpdated* when the graph is modified preserves UPDATED. The rest follows from *removeVertex* of the *GraphRepresentation* class. ■

The *addEdge* method takes *e*, the edge to add to this graph. This method requires that multi-edges are added only to multigraphs. This requirement is checked by an **assert statement**. The advantage of using an assert statement is that the check is performed only when the Java virtual machine is launched with the argument that specifies assertions are enabled.

```

public void addEdge(E e) {
    assert allowsMultiEdges() || !graph.containsEdge(e.source(), e.dest());
    graphUpdated();
    graph.addEdge(e);
}

```

Correctness Highlights: By the correctness of the *allowsMultiEdges* method and the *containsEdge* method of the *GraphRepresentation* class, the assertion only fails when there is an attempt to add a multi-edge when it is not allowed. The call to *graphUpdated* when the graph is modified preserves UPDATED. The rest follows from *removeEdge* of the *GraphRepresentation* class. ■

The *removeEdge* method takes *e*, the edge to remove. It returns *true* if *e* is removed, or *false* if *e* is not in the graph.

```

public boolean removeEdge(E e) {
    graphUpdated();
    return graph.removeEdge(e);
}

```

Correctness Highlights: The call to *graphUpdated* when the graph is modified preserves UPDATED. The rest follows from *removeEdge* of the *GraphRepresentation* class. ■

The *edgeSet* method returns a set of all edges in this graph.

```
Set<E> edgeSet() {
    Set<E> set = new OpenAddressing<E>();
    for (V v: graph) {
        Iterator<E> edges = edgesFrom(v);
        while (edges.hasNext())
            set.add(edges.next());
    }
    return set;
}
```

The *iterator* method returns a non-mutating iterator defined over all vertices that is initialized to be just before the first vertex in the set. This iterator is non-mutating to ensure that the application program can only remove a vertex through the *removeVertex* method, which preserves UPDATED.

```
public NonMutatingIterator<V> iterator() {
    return graph.iterator();
}
```

The *edgesFrom* method takes *v*, the desired vertex. It returns an iterator defined over all outedges of *v* that is initially positioned just before the first one. If the graph is an undirected graph, then the iteration order will include all edges incident to *v*.

```
public Iterator<E> edgesFrom(V v) {
    return graph.edgesFrom(v);
}
```

The *edgesTo* method takes *v*, the desired vertex. It returns a non-mutating iterator defined over all inedges of *v* that is initialized just before the first one. If the graph is an undirected graph then the iterator will include all edges incident to *v*. This method throws an *UnsupportedOperationException* when the graph representation does not support this capability.

```
public Iterator<E> edgesTo(V v) {
    if (!isDirected())
        return edgesFrom(v);
    return graph.edgesTo(v);
}
```

53.3 In-Tree Representation

TaggedCollectionWrapper<V,TreeData<V,E>> implements *TaggedCollection*<V,TreeData<V,E>>

└ OpenAddressingMapping<V, TreeData<V,E>> implements *Mapping*<V, TreeData<V,E>>

└ InTree<V, E extends Edge<V>> implements *Mapping*<V, TreeData<V,E>>

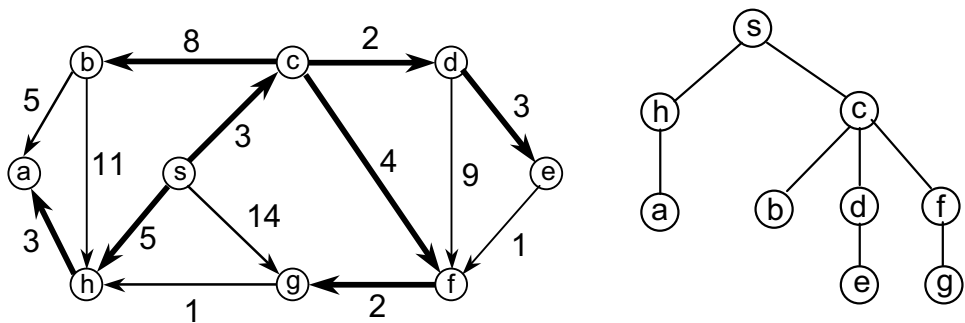


Figure 53.1
On the left is a directed weighted graph, with the edges that form a shortest path from s to some other vertex shown in bold. On the right is the shortest path tree for this weighted graph.

An important problem for both unweighted and weighted graphs is finding a shortest path between two vertices. For an unweighted graph, the length of a path is defined as the number of edges in the path. For a weighted graph, the length of the path can be measured either in terms of the number of edges along the path (an **unweighted path**) or in terms of the sum of the edge weights along the path (a **weighted path**).

Recall that in the single-source shortest path problem, the goal is to compute a shortest path from a specified source vertex to each other vertex. The correct algorithm choice for the single-source shortest path problem depends on whether the graph is unweighted, weighted with non-negative weight edges, or weighted with arbitrary (possibly negative) weight edges.

All of these algorithms compute a shortest path tree that stores within it the shortest paths from the source to all other vertices in the graph. Figure 52.2 shows a shortest path tree for an unweighted graph, and Figure 53.1 shows a shortest path tree for a weighted graph. An edge in the shortest path tree from parent u to child v corresponds to an edge (u, v) in the graph. Although the shortest path tree is constructed to represent a path from s to an arbitrary vertex v , directly representing the child relationship is problematic since each vertex can have up to n children. In contrast, every vertex in the shortest path tree has a unique parent. Furthermore, by traversing the parent pointers from v up to s , the path from s to v can be constructed. So an in-tree (Section 1.6) in which each vertex stores only its parent is the best representation the solution to this problem. Therefore, the single source shortest path problem reduces to the problem of constructing an in-tree where the root is the source and each path from a vertex to the root represents the shortest path from the root to that vertex.

While algorithms that solve the single-source shortest path problem are the only ones that return a shortest path tree, there are other algorithms that use the InTree class. For example, Prim's minimum spanning tree algorithm (Section 57.3) internally uses an in-tree to store edges that are being considered for placement in the minimum spanning tree. An in-tree is also used within the ShortestPathMatrix (Section 57.6), which stores a shortest path between all pairs of vertices.

Typically, an in-tree stores a parent vertex for each vertex. However, suppose u is the parent of v in the in-tree for multigraph G . There could be many distinct edges from u to v in G . Thus, it is not sufficient to simply record that u is the parent of v since that would not identify a unique edge. To resolve this problem, our implementation for an in-tree stores, for each vertex v , the edge e from v 's parent to v . Observe that the only additional cost of storing e , as opposed to v 's parent, is that one extra method call is required to retrieve v 's parent from the edge.

A pathological case arises for the shortest path problem when a weighted graph contains a negative weight cycle. Consider a weighted graph in which vertex a is reachable from the source and there is an edge of weight 5 from a to b , an edge of weight 3 from b to c , and an edge of weight -9 from c to a . The sum of the edge weights on the cycle $a \rightarrow b \rightarrow c \rightarrow a$ is -1 . By going around

this cycle many times, the weight of the resulting path can be made arbitrarily small. In such a case, the shortest path from s to any reachable vertex is ill defined. When this situation occurs, Bellman-Ford's single-source shortest path algorithm (Section 57.5) is guaranteed to return an in-tree that represents some negative weight cycle (using the parent references).

In-Tree Data

The in-tree is stored as a mapping between the vertices in the graph and `TreeData` instances, where the `TreeData` instance for vertex v holds *edgeFromParent* that defines v 's parent in the in-tree, and *cost*, the cost of the path represented in the in-tree from s to v .

```
class TreeData<V, E> {
    E edgeFromParent;
    double cost;

    TreeData(E e, double cost) {
        edgeFromParent = e;           //edge from parent in in-tree from source
        this.cost = cost;              //distance of in-tree path from source
    }
}
```

Instance Variables and Constants: The `InTree` class extends `OpenAddressingMapping` (Section 49.7.2). It adds two instance variables, the source vertex, and *cycleStart*, a vertex that is part of a negative weight cycle in the graph reachable from the source, or *null* if there is no negative weight cycle in the graph reachable from the source.

```
V source;
V cycleStart = null;
```

In-Tree Methods

The constructor takes *source*, the source vertex that is the root of the in-tree.

```
InTree(V source) {
    this.source = source;
}
```

Vertices are placed into the in-tree with the *add* method that takes v , the vertex to add to the tree, *edgeFromParent*, the edge to place in the path that defines the parent of v , and *cost*, the cost associated with v with respect to the specified parent. This method returns the tree data instance that is associated with v .

```
TreeData<V, E> add(V v, E edgeFromParent, double cost) {
    TreeData<V, E> data = new TreeData<V, E>(edgeFromParent, cost);
    put(v, data);           //create an association between v and data
    return data;
}
```

The *getSource* method returns the source vertex for this in-tree.

```

public V getSource() {
    return source;
}

```

The *isReachableFromSource* method takes *dest*, the destination vertex. It returns *true* if and only if there is a path in this tree from the source to *dest* in this graph.

```

public boolean isReachableFromSource(V dest) {
    if (source == dest)                                //any vertex is reachable from itself
        return true;
    return contains(dest) && get(dest).edgeFromParent != null;
}

```

The *getPathDistanceFromSource* method takes *dest*, the destination vertex. It returns the cost of the path represented in the in-tree from the source to *dest*. By definition, an unreachable vertex has infinite cost. When *dest* is not reachable from the source, infinity is returned.

```

public double getPathDistanceFromSource(V dest) {
    if (source == dest)
        return 0;
    else if (!contains(dest))
        return AbstractGraph.INF;
    else
        return get(dest).cost;
}

```

The *getPathFromSource* method takes *dest*, the destination vertex. It returns a positional collection containing the edges along the path from the root to *dest*, in order. If *dest* is not reachable from *s*, *null* is returned.

```

public PositionalCollection<E> getPathFromSource(V dest) {
    if (!isReachableFromSource(dest))                    //no path from the source to dest
        return null;
    CircularArray<E> path = new CircularArray<E>(getSize());
    V curVertex = dest;
    while (curVertex != source) {
        E e = get(curVertex).edgeFromParent;
        path.addFirst(e);
        curVertex = AbstractGraph.neighbor(curVertex, e);
    }
    return path;
}

```

Correctness Highlights: Since the in-tree's parent edges are directed toward the root, starting at *dest* and following the parent references yields the desired path (in reverse order). Inserting the current edge to the front of the path places the edges in the order in which they occur from the source to *dest*. ■

In the normal case, the in-tree has no cycles. However, when the algorithm building the in-tree detects that the problem is ill-defined for the graph on which it was called due to cycle *C*, then the “in-tree” data structure is modified to represent *C*. At that point the “in-tree” can no longer be thought of as a tree.

The *hasNegativeWeightCycleReachableFromSource* method returns *true* if and only if there is a negative weight cycle reachable from *s*.

```
public boolean hasNegativeWeightCycleReachableFromSource() {
    return cycleStart != null;
}
```

The internal *buildCycleStartingAt* method takes *start*, a vertex that is guaranteed to be part of negative weight cycle in the graph. It returns a positional collection containing the edges, in order, in a negative weight cycle in the graph, where the cycle includes *start*. This method requires that there is a cycle in the in-tree that includes *start*.

```
PositionalCollection<E> buildCycleStartingAt(V start) {
    CircularArray<E> directedCycle = new CircularArray<E>(getSize());
    V u = get(start).edgeFromParent.source();
    while (u != start) {                                     //follow parent until start is reached
        E e = get(u).edgeFromParent;
        directedCycle.addFirst(e);
        u = e.source();
    }
    directedCycle.addLast(get(start).edgeFromParent);
    return directedCycle;
}
```

The public *getCycleReachableFromSource* method returns a positional collection containing the edges of a cycle in the graph reachable from the source vertex, or *null* if there is no cycle in the graph reachable from the source.

```
public PositionalCollection<E> getCycleReachableFromSource() {
    if (cycleStart == null)
        return null;
    else
        return buildCycleStartingAt(cycleStart);
}
```

53.4 Finding Shortest Paths with Breadth-First Search

In this section, we present an implementation of **breadth-first search (bfs)** to compute the shortest path tree in an unweighted graph for a specified source vertex *s*. The basic idea of breadth-first search is as follows. The shortest path tree *T* is initialized to be the singleton root *s*. Let $V_i \subseteq V$ denote the set of vertices that have a *shortest* path from *s* that use *i* edges. By definition, $V_0 = \{s\}$, and V_1 is the set of vertices reachable from *s* by one edge. More formally,

$$V_1 = \{v \in V - V_0 \mid (s, v) \in E\}.$$

Clearly, the shortest path from *s* to vertex $v \in V_1$ is the path $\langle s, v \rangle$ of length 1. So each $v \in V_1$ is placed as a child of *s* in *T*. Since all vertices reachable from *s* by a single edge are in *T*, all remaining vertices have a shortest path of at least two edges. Next all vertices not in *T* that are reachable from some $u \in V_1$ are added to *T*. More specifically for each outgoing edge (u, v) where $u \in V_1$ and $v \notin T$, *v* is added to *T* as a child of *u*. When *v* is added to *T* based on the edge (u, v) ,

we say that v is **discovered** by u . Combining (u, v) with the parent edge (s, u) for u , yields the path $s \rightarrow u \rightarrow v$ of length 2. Since all vertices not in $V_0 \cup V_1$ have a shortest path of length at least 2, this is a shortest path to v . So each vertex in V_2 is correctly added to T during this step. Next all outgoing edges from vertices in V_2 are considered to add all vertices in V_3 to T , and so on. This process stops only after all outgoing edges have been considered for all vertices in T , and no new vertices are discovered.

To implement breadth-first search, an efficient way is needed to iterate over the outgoing edges in V_0 , then V_1 , then V_2 , and so on. Since all vertices in V_i are discovered before any vertex in V_{i+1} , we can achieve this goal by placing the vertices in a queue (Chapter 18) when they are discovered. By visiting the outgoing edges of the vertices, in the order they were placed in the queue, all outgoing edges from a vertex in V_i are considered before any edge from a vertex in V_{i+1} .

We now describe our breadth-first search implementation. When v is not reachable from s , then the shortest path has infinite length. We use the constant *INF* for the length of the shortest path from s to any unreachable vertex.

```
public static final double INF = java.lang.Double.POSITIVE_INFINITY;
```

Throughout this section we use $d(v)$ to denote the tree data instance associated with v . That is, $d(v) = \text{tree.get}(v)$. So the path defined by the shortest path tree from the source s to v is the path obtained by repeatedly following *edgeFromParent* and adding that edge to the front of the path until s is reached.

The correctness of breadth first search is proven by showing that the following properties are maintained by *bfs(s)*.

DISTANCE: For any vertex v , if v is in the shortest path tree, then $d(v).dist$ is the number of edges in a shortest path tree from s to v .

SHORTESTPATH: Whenever vertex v is added into the shortest path tree, the path from s to v defined by the shortest path tree is a shortest path from s to v .

QUEUE: The vertices are placed in the queue in non-decreasing order based on the distance of their shortest path from s . More formally, let $\langle q_1, q_2, \dots, q_k \rangle$ be the vertices in the queue with q_1 at the front of the queue and q_k at the end of the queue. For $i \in \{1, \dots, k-1\}$, $d(q_i).dist \leq d(q_{i+1}).dist$.

REACHABLE: A vertex is added to the shortest path tree and the queue if and only if it is reachable from the source.

The *unweightedShortestPath* method takes s , the source vertex. It returns an in-tree that is a shortest path tree for source vertex s . The shortest path tree returned is also called a **breadth-first search tree**.

```
public InTree<V, E> unweightedShortestPaths(V s) {                                     //uses bfs
    InTree<V, E> shortestPathTree = new InTree<V, E>(s);
    shortestPathTree.add(s, null, 0);
    Queue<V> q = new Queue<V>();                                                    //create an empty queue of vertices
    q.enqueue(s);                                                                    //place source in the queue
    while (!q.isEmpty()) {                                                            //while some vertex is still in the queue
        V u = q.dequeue();                                                            //u is next vertex in the queue
        TreeData<V, E> uData = shortestPathTree.get(u);
        Iterator<E> outEdges = edgesFrom(u);
        while (outEdges.hasNext()) {
```

```

    E e = outEdges.next();           //e is edge from u to v
    V v = neighbor(u, e);           //v is reachable from u by edge e
    if (!shortestPathTree.contains(v)) { //v is discovered
        shortestPathTree.add(v, e, uData.cost + 1);
        q.enqueue(v);               //put v at the end of the queue
    }
}
}
return shortestPathTree;
}

```

Correctness Highlights: First we argue that the four properties hold before the while loop begins. The only vertex in the shortest path tree and the queue is s . Since s is the root, its *parentEdge* should be *null*, and by definition, its distance to itself is 0. So DISTANCE and SHORTESTPATH hold. Since the queue only holds one vertex, QUEUE vacuously holds. Finally, s is reachable from itself, and so REACHABLE holds.

We now argue that all four properties are preserved by the while loop. Let $T \subseteq V$ be the set of vertices that have been placed in the shortest path tree. Before the while loop begins, $T = \{s\}$. By QUEUE and DISTANCE, among all vertices in $V - T$, u is a vertex with a shortest path from s . Let edge $e = (u, v)$ be the edge considered by the inner while loop. By SHORTESTPATH if v is in the shortest path tree, then the shortest path from s to v has already been found.

We now consider when v is not in the shortest path tree. By DISTANCE, $uData.dist$ is the length of the shortest path from s to v , and by SHORTESTPATH, the path P defined by the shortest path tree from s to v has distance $uData.dist$. Clearly, appending the edge (u, v) to P yields a path from s to v of length $uData.dist + 1$, and by setting *parentEdge* for v to e , this path is the one defined by the shortest path tree for v . By QUEUE, v was not reachable from any vertex with shortest distance less than $uData.dist$, and so this path to v must be a shortest path. Thus DISTANCE, SHORTESTPATH, and QUEUE are preserved. Finally, REACHABLE is preserved since v is only placed into the shortest path tree, and in the queue, when a path from s to v is found.

When the while loop terminates, all properties will hold and the queue is empty. Thus, all outgoing edges from all vertices in the shortest path tree have been examined, and no new vertices have been discovered. Thus, all reachable vertices are in the shortest path tree, and the shortest path from each reachable vertex v has length $v.cost$, and is represented in the tree returned. ■

53.5 Finding Cycles and Connected Components with Depth-First Search

We now describe **depth-first search (dfs)**, which is an alternate way to search a graph to find all reachable vertices. While depth-first search does not find shortest paths, it can be used to find a cycle in a directed graph, to find the connected components of a graph, to find a valid topological order in an acyclic graph, and to compute the strongly connected components of a directed graph. As discussed in the case study of Section 53.9, an in-place version of depth-first search is commonly used by garbage collection algorithms to mark all of the cells that are reachable (and thus not garbage).

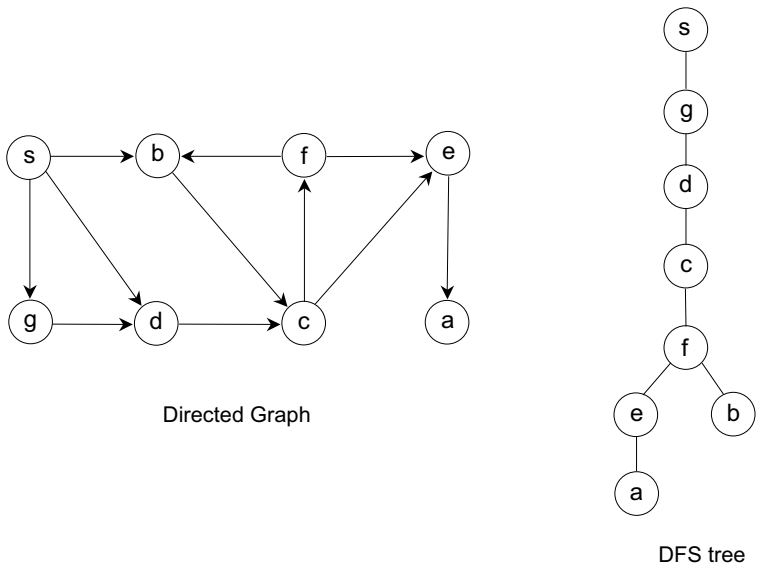


Figure 53.2
The same directed graph from Figure 52.2, and the depth-first search tree obtained when the vertices are visited in reverse alphabetical order. This order places *s* at the root.

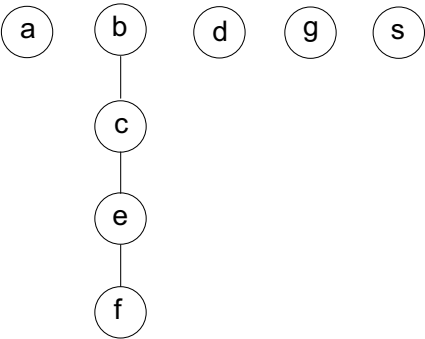


Figure 53.3
The depth-first search forest that is obtained when visiting the vertices in the graph of Figure 53.2 in alphabetical order.

At a high-level, depth-first search differs from breadth-first search only in that it uses a stack (Chapter 19) instead of a queue to determine the order in which to process the vertices that have been discovered. Figure 53.2 shows the **depth-first search tree** for the same graph as that of Figure 52.2 (which shows the breadth-first search tree). Depth-first search starts at s and visits the outedges from each vertex. In this example, we assume the children are visited in reverse alphabetical order based on the destination. Unlike breadth-first search that would visit g , d , and then b (if using reverse alphabetical order), when depth-first search visits g , g is placed on the top of the stack. Thus before continuing to visit the other vertices reachable from s , the outgoing edges of g are considered. When d is discovered it is put on the stack on top of g , which is on top of s . Then the outgoing edges from d are considered. The choice of the names breadth-first search and depth-first search becomes clear upon comparing the shapes of the breadth-first search tree and the depth-first search tree. In breadth-first search, all elements reachable are considered, and then all elements reachable from those are considered, and so on, generally leading to a wide shallow tree. In contrast, depth-first search keeps going deeper and deeper into the search until no new vertices can be reached. Thus, in general, depth-first search leads to taller trees where the number of children per vertex (especially towards the root) tends to be smaller.

The most significant difference between depth-first search and breadth-first search is the use of a stack instead of a queue. Although the Stack class (Chapter 19) could be used for depth-first search, we instead illustrate the more standard approach of using recursion, where the stack is Java's execution stack. The other differences between depth-first search and breadth-first search relate to the way in which these two algorithms are used. Since breadth-first search is used to solve the single-source shortest path problem from a specified source, the vertices unreachable from the source are never discovered. In contrast, for depth-first search, typically there is no specified source vertex. Instead, a top-level depth-first search procedure iterates through all vertices in the graph. Whenever an undiscovered vertex v occurs during this iteration, a recursive method $dfsVisit(v)$ is called. When a vertex v is placed on the stack, we say that vertex v is **visited**. The call to $dfsVisit(v)$ will recursively visit any vertex reachable from v that has not been visited in an earlier call to $dfsVisit$. Thus, in general, depth-first search creates a **depth-first search forest** that can be composed of up to n trees. Figure 53.3 shows the depth-first search forest created by dfs for the directed graph of Figure 53.2, assuming that both the top-level iteration and the iteration over the outedges of each vertex are alphabetical.

The final difference between depth-first search and breadth-first search is the data they associate with each vertex. As with breadth-first search, the depth-first search forest is represented by storing *edgeFromParent* for each vertex. For each root in the forest, *edgeFromParent* is *null*. Since the number of edges from s to v in the depth-first search tree has no significance, there is no need to retain this value. However, depth-first search can be used to detect and build a directed cycle. In order to perform this task, it is important to store whether all of the outgoing edges of each vertex have been considered. We say that a vertex v is **finished** when this occurs. Operationally, vertex v is finished when $dfsVisit(v)$ completes, or equivalently when v is removed from the stack.

Recall that a connected component of an undirected graph is a maximal set of vertices that are mutually reachable from each other. It is easily shown that each tree in the depth-first search forest corresponds to a connected component in an undirected graph.

DFSData Inner Class

The DFSData class is used to hold the information associated with each vertex during depth-first search.

```
static class DFSData<E> {
    E edgeFromParent;           //edge to parent in the DFS forest
    boolean finished;           //true when the vertex has been removed from the stack
}
```

The *DFSData* constructor takes *e*, an edge used to initialize the value of *edgeFromParent*. Since the *DFSData* instance for a vertex is created when the vertex is first placed on the stack, *finished* is initially false.

```
public DFSData(E e) {
    finished = false;
    edgeFromParent = e;
}
```

The factory method *newDFSData* takes *v*, the vertex to be associated with this *DFSData* instance, and *e*, the edge that leads to *v* in the shortest path tree (i.e., the parent edge for *v*).

```
DFSData<E> newDFSData(V v, E e) {
    DFSData<E> data = new DFSData<E>(e);
    dfsForest.put(v, data);
    return data;
}
```

Depth-First Search Methods

The *dfsUpdateAsNeeded* method calls *dfs* if the graph has been updated since last time *dfs* was run.

```
void dfsUpdateAsNeeded() {
    if (topOrder == null) {
        dfsForest = new OpenAddressingMapping<V, DFSData<E>>(numVertices());
        topOrder = new CircularArray<V>(numVertices());
        components = new OpenAddressing<Set<V>>();
        dfs();
    }
}
```

Correctness Highlights: By UPDATED, when *topOrder* is null, *dfs* has not been run since the graph was created or updated. Resetting *dfsForest*, *topOrder*, and *components* just before *dfs* is executed preserves UPDATED. ■

The *dfsVisit* method uses the following method to determine if any edge encountered creates a cycle. The *checkForCycle* method takes *u*, a possible “starting” point for a directed cycle in this graph, and *e* = (*u*, *v*), an outgoing edge of *u* in the potential cycle. If *v* is not finished, and we are processing *u* in a *dfs* recursive call, then there must be a path from *v* to *u*. Combined with *e*, a cycle is formed. So, if *v* is not finished, this method constructs a directed cycle and stores it in *directedCycle*. Otherwise, the method does nothing. This method requires that *u* is at the top of the *dfs* stack.

```
void checkForCycle(V u, E e) {
    V v = e.dest();
    DFSData<E> vData = dfsForest.get(v);
    if (!vData.finished) { //this implies there is a cycle
        directedCycle = new CircularArray<E>(numVertices());
    }
}
```

```

    while (u != v) {
        E nextEdge = dfsForest.get(u).edgeFromParent;
        directedCycle.addFirst(nextEdge);
        u = neighbor(u, nextEdge);
    }
    directedCycle.addLast(e);
}
}

```

Correctness Highlights: We argue that if the conditional is entered then *directedCycle* will hold a directed cycle in this graph. By the requirement on this method, *u* is at the top of the recursion stack. Since $e = (u, v)$ and *v* is not finished when the conditional is entered, it implies that *v* is somewhere in the recursion stack (deeper than *u*). However, for *u* to be at the top of the stack containing *v*, a path from *v* to *u* must have been used to discover *u*. Thus, *v* is an ancestor of *u* in *dfsForest*. This implies that by following the *edgeFromParent* from *v*, eventually *u* must be reached. By reversing the order of the parent edges followed, and also placing *e* last in *directedCycle*, when the while loop terminates *directedCycle* holds a cycle in the graph. ■

The recursive *dfsVisit* method takes *u*, the vertex to visit next, *uData*, the DFSData instance associated with *u*, and *component*, an empty set in which to place the vertices of the connected component. This method visits all vertices reachable from *u* that are not yet in *dfsForest*. The DFSData instance associated with *u* is included as a parameter as an optimization to avoid retrieving it a second time. This method requires that *component* is *null* when the graph is undirected.

```

void dfsVisit(V u, DFSData<E> uData, Set<V> component) {
    if (component != null)                                //if an undirected graph
        component.add(u);                                //place newly discovered vertices into component
    Iterator<E> outEdges = edgesFrom(u);                  //iterate over all outedges of u
    while (outEdges.hasNext()) {
        E e = outEdges.next();
        if (e != uData.edgeFromParent) {                  //don't use same edge in both directions
            V v = neighbor(u, e);                          //v is reachable from u
            if (!dfsForest.contains(v))                    //if v is not yet in dfsForest
                dfsVisit(v, new DFSData(v, e), component); //put v in dfsForest and visit
            else if (directedCycle == null)                 //check for a directed cycle if none found yet
                checkForCycle(u, e);
        }
    }
    uData.finished = true;                                //finished exploring outedges from u
    topOrder.addFirst(u);                                  //add u to front of the topological order
}

```

Correctness Highlights: Since the connected components are defined only for an undirected graph, it would be wasteful to construct *components* for a directed graph. Thus *component* is only allocated when this graph is undirected. So if *component* is not *null*, then each vertex should be inserted into it when it is visited.

The while loop considers each outgoing edge from *u*. Let $e = (u, v)$ be the edge being considered. Observe that if *uData.edgeFromParent* is *e*, then *e* is an undirected edge used to reach *u* from *v*. Thus, *v* must be in *dfsForest* and need not be considered. Along with a small

savings in computation, not considering such an edge further, prevents a path from u to v to u that uses edge e twice from being treated as a cycle in an undirected graph.

Observe that any vertex v discovered from u in $dfsVisit(u, uData)$ is placed in $dfsForest$ with an edge $e = (u, v)$ as the *parentEdge* for v . Then v is recursively visited. Thus, $dfsVisit(u, uData)$ will recursively visit any vertex reachable from u before it terminates. Also any vertex added to the front of *topOrder* exactly when all vertices reachable from it already have been visited recursively.

Observe that the requirements of *checkForCycle* are met since $e = (u, v)$ is in the graph, and u is at the top of the recursion stack (since the call is made by *dfsVisit* on vertex u). Thus, if v is not finished, by the correctness of *checkForCycle*, the global variable *directedCycle* will hold a directed cycle. ■

The **top-level dfs** method iterates over all vertices, visiting each one that is not yet in the depth-first search forest. Observe that a vertex is placed into the depth-first search forest by *newDFSData* just before it is visited (i.e., placed on the “stack”).

```
void dfs() {
    Set<V> component = null;
    for (V u: this) { //iterate through all vertices in the graph
        if (!dfsForest.contains(u)) { //if u is not yet in forest
            if (!isDirected()) //if the graph is undirected
                component = new OpenAddressing<V>(); //found a new component
            dfsVisit(u, newDFSData(u, null), component); //place u in forest and visit
        }
        if (!isDirected()) //if the graph is undirected
            components.add(component); //add new component
    }
    dfsForest = null; //free space
}
```

Correctness Highlights: By the correctness of *dfsVisit*, the Mapping *contains* method, and the *newDFSData* constructor, each vertex $v \in V$ is in *dfsForest* with *v.edgeFromParent* being an edge with destination v .

Observe that any element placed in *dfsForest* by this method was not reachable from any vertices finished. Furthermore, for an undirected graph, by the correctness of *dfsVisit*, any vertices reachable from u will be discovered and placed into *component* which has u as its root in the depth-first search forest. Thus, if the graph is undirected, *components* will hold a set of vertices for each component of the graph.

By the correctness of *dfsVisit*, when *dfs* terminates, all vertices will be in *topOrder*, in the reverse order of when they are finished (i.e., removed from the stack).

We now argue that if there is a cycle in this graph, then at some point *checkForCycle* is called by *dfsVisit* for a vertex v that is not finished. Suppose not. Let v be the first vertex discovered in some cycle C of the graph (either by *dfs* or *dfsVisit*). Let u be the vertex that precedes v in C . Since *dfsVisit* for v cannot finish until all vertices reachable from v have been visited, and u is reachable from v , at some point *dfsVisit* for u is called when v is somewhere in the recursion stack. Given that no cycle has yet been found, when the outgoing edge (u, v) from u is considered in *dfsVisit*, *checkForCycle* is called for v which by definition is unfinished. Thus by the correctness of *checkForCycle*, this cycle is correctly placed in *directedCycle*. Thus, if a directed graph is cyclic, then *dfs* will set *directedCycle* to hold a directed cycle in the graph.

Observe that *dfsVisit* prevents the same edge from being used in both directions in an undirected graph by only considering an edge $e = \{u, v\}$ during *dfsVisit*(*u*) when *e* is not the parent edge for *u*. ■

The *hasCycle* method returns *true* if and only if this graph has a directed cycle.

```
public boolean hasCycle() {
    dfsUpdateAsNeeded();
    return directedCycle != null;
}
```

Correctness Highlights: Follows from UPDATED and the correctness of *dfs*. ■

The *getCycle* method returns a positional collection holding a directed cyclic in the graph. It throws a *GraphException* when the graph is acyclic. The *hasCycle* method can be used to determine if the graph has a directed cycle.

```
public PositionalCollection<E> getCycle() {
    dfsUpdateAsNeeded();
    if (directedCycle != null)
        return directedCycle;
    else
        throw new GraphException("no directed cycle exists");
}
```

The *numConnectedComponents* method returns the number of connected components in this graph. It throws a *GraphException* when the graph is directed, since the connected components are defined only for undirected graphs.

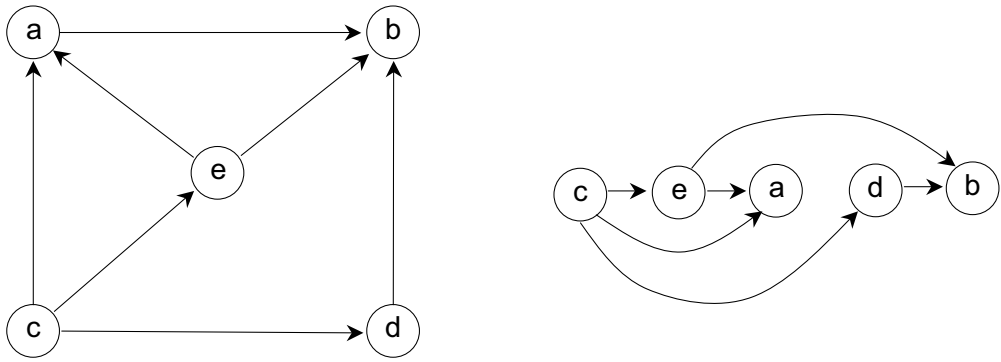
```
public int numConnectedComponents() {
    if (isDirected())
        throw new GraphException("applies only to undirected graphs");
    dfsUpdateAsNeeded();
    return components.getSize();
}
```

The *getConnectedComponents* method returns the connected components in this graph. It throws a *GraphException* when the graph is directed, since the connected components are only defined for undirected graphs.

```
public Set<Set<V>> getConnectedComponents() {
    if (isDirected())
        throw new GraphException("applies only to undirected graphs");
    dfsUpdateAsNeeded();
    return components;
}
```

53.6 Topological Sort

As discussed in Section 52.1, some applications involve finding an order in which to perform a set of tasks such that the order obeys a set of precedent constraints. Recall that an ordering of the tasks

**Figure 53.4**

A sample acyclic directed graph is shown on the left. On the right the vertices are placed along a line in a valid topological order. Observe that when drawn this way, all edges in the graph are directed towards a vertex later in the ordering. In general, there may be several valid topological orders. In this example, *d* could be placed anywhere between *c* and *b*.

that does not violate any of the precedence constraints is called a topological order. This name is derived from the fact that if the vertices in the graph placed along a line according to the topological order, then all of the edges are directed from the left to the right. Figure 53.4 shows a small graph drawn in this manner. Knuth [96] observed that the reverse order of when the vertices are finished (i.e., removed from the stack) in depth-first search is a valid topological order when the graph is acyclic. We prove this fact in the correctness argument for *topologicalOrder*.

The *topologicalOrder* method returns a positional collection holding a valid topological order in the graph. It throws a *GraphException* when the graph is cyclic. The *hasCycle* method can be called first to find out if the graph has a cycle.

```
public PositionalCollection<V> topologicalOrder(){
    dfsUpdateAsNeeded();
    if (directedCycle == null)
        return topOrder;
    else
        throw new GraphException("a directed cycle exists");
}
```

Correctness Highlights: By *UPDATED* and the correctness of *dfs*, an exception is properly thrown if the graph is cyclic, and otherwise *topOrder* holds the permutation of the vertices in the graph defined by the reverse order of the finishing time. We now prove that *topOrder* is a valid topological order when the graph is acyclic. Suppose not. Then vertex *v* must precede vertex *u* in *topOrder*, when the edge (u, v) is in the graph. That is, *v* must finish after *u*. We consider the following two cases.

Case 1: *v* not discovered when *u* is finished. By the correctness of *dfsVisit*, all vertices reachable from *u* are discovered by the completion of *dfsVisit* on *u*. Since *v* is not discovered when *u* is finished, then *v* cannot be reachable from *u*, which contradicts that the edge (u, v) is in the graph.

Case 2: *v* is on the stack (but not finished) when *u* is finished. Thus *v* must be deeper in the stack than *u*. As discussed in the correctness proof for *dfs*, if *v* is deeper in the stack than *u*,

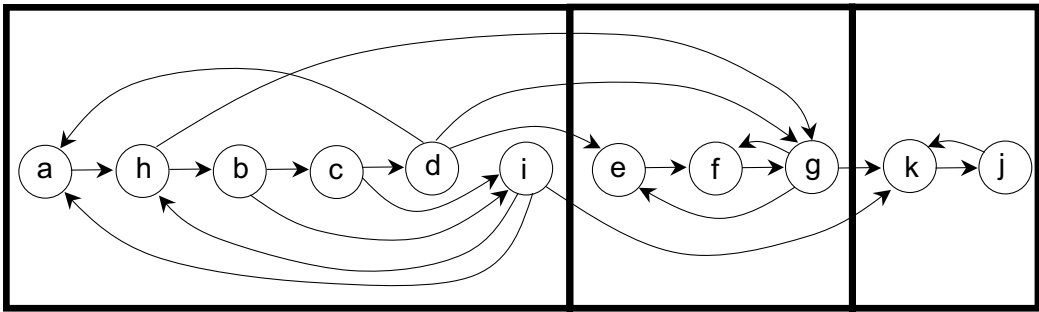


Figure 53.5

A sample cyclic graph is shown with the vertices in the reverse order of when they are finished by *dfs* when the top-level iteration and all iterators of the outedges consider the vertices in alphabetical order. The rectangles enclose each of the three strongly connected components in this graph.

then there must be a path from v to u in the graph. Otherwise, v would have finished before u was discovered. So there must be a path from v to u in the graph. Combined with the edge (u, v) this implies there is a cycle in G , contradicting the fact that the graph is acyclic.

A contradiction is reached in both cases. If the graph is acyclic and v precedes u in *topOrder*, then there is no edge from u to v in the graph. Thus *topOrder* is a valid topological order. ■

53.7 Strongly Connected Components

Another application of depth-first search is to compute the strongly connected components of a directed graph. Recall that a **strongly connected component** is defined as a maximum-sized set of vertices such that every two vertices in a component are reachable from each other. Figure 53.5 shows the strongly connected components of a sample directed graph.

The strongly connected component algorithm begins by applying depth-first search to the graph. After completing the first depth-first search, the strongly connected components algorithm performs a second depth-first search with the added restriction that depth-first search iterates over the vertices in the order of their finishing times from the initial depth-first search. (Recall that *topOrder* holds the vertices in *reverse order* of finishing time.) Finally, each tree in the depth-first search forest from the second execution of depth-first search is a strongly connected component.

Let $G = (V, E)$ be the original graph. A **vertex contraction** of a graph combines two or more vertices into one, removing any edges between the vertices being contracted. To develop intuition about why the strongly connected components algorithm is correct, we define a **contracted graph** $G^* = (V^*, E^*)$ in which there is one vertex in V^* for each strongly connected component of G , and there is an edge in E^* between vertices u^*, v^* , exactly when there exists a vertex $u \in u^*$ and a vertex $v \in v^*$ for which $(u, v) \in E$. For example, each rectangle in Figure 53.5 corresponds to a vertex in G^* . There would be three edges in E^* , one from the leftmost rectangle to the middle rectangle, one from the middle rectangle to the rightmost rectangle, and one from the leftmost rectangle to the rightmost rectangle. By the definition of a strongly connected component, each component is a maximal set of vertices that are mutually reachable. Thus, G^* must be acyclic. (If there were a cycle between two vertices, then all vertices among the two components would be mutually reachable, making them a single component.) Intuitively, the first execution of depth-first search places the vertices in V^* into a valid topological order, $V_1^*, \dots, V_k^*$. Since the second depth-

first search starts at the last vertex in *topOrder*, it starts at some vertex in V_k^* . So the first call to *dfs-visit* will reach exactly those vertices in V_k^* , making them a tree in the depth-first search forest. The next undiscovered vertex moving backwards through *topOrder* is some vertex in V_{k-1}^* . Then *dfsVisit* for that vertex will discover exactly those vertices in V_{k-1}^* , and so on.

The *sccUpdateAsNeeded* method calls *stronglyConnectedComponents* to update *scc* if the graph has been updated since last time *stronglyConnectedComponents* was run.

```
void sccUpdateAsNeeded() {
    if (scc == null) {
        scc = new OpenAddressing<Set<V>>();
        stronglyConnectedComponents();
    }
}
```

Correctness Highlights: By UPDATED *scc* is only null when *stronglyConnectedComponents* has not been run since the graph was created or updated. Setting *scc* to an empty set just before calling *stronglyConnectedComponents* preserves UPDATED. ■

The internal *dfsVisitForSCC* method takes *u*, the vertex from which to start, *uData*, the data associated with *u*, and *component*, a set that is to hold all elements in this strongly connected component. This method is just like *dfsVisit* except that it does not modify *topOrder*, and it adds each vertex visited in this directed graph into *component*. It requires that this is a directed graph.

```
void dfsVisitForSCC(V u, DFSData<E> uData, Set<V> component) {
    component.add(u);           //place all newly discovered vertices into this component
    Iterator<E> outEdges = edgesFrom(u);
    while (outEdges.hasNext()) {
        E e = outEdges.next();
        if (!dfsForest.contains(e.dest()))
            dfsVisitForSCC(e.dest(), newDFSData(e.dest(), e), component);
    }
    uData.finished = true;      //finished exploring inedges to u
}
```

The *stronglyConnectedComponents* method computes the strongly connected components of the graph storing the result in *scc*. The application program can access the strongly connected components through either *numStronglyConnectedComponents* or *getStronglyConnectedComponents*.

```
void stronglyConnectedComponents() {
    dfsUpdateAsNeeded();       //if needed, run depth-first search on the original graph
    dfsForest = new OpenAddressingMapping<V, DFSData<E>>(numVertices()); //reset
    Set<V> component = null;
    for (int p = numVertices()-1; p ≥ 0; p--) {           //in finishing order from first dfs
        V u = topOrder.get(p);
        if (!dfsForest.contains(u)) {                     //visit u if not yet in dfsForest
            component = new OpenAddressing<V>();
            dfsVisitForSCC(u, newDFSData(u, null), component);
        }
        scc.add(component);
    }
}
```

```

    }
    dfsForest = null;
}

```

Correctness Highlights: Let $V_0, \dots, V_{k-1}$ be the strongly connected components, and let $n = \text{numVertices}()$. For the contracted graph G^* described above, suppose that the topological order of the vertices in G^* is $\langle V_0, \dots, V_{k-1} \rangle$. We now argue that after the completion of this method, *scc* contains one set for each V_i , containing the vertices in strongly connect component V_i .

As proven above, G^* is acyclic, so the initial call to *dfs* computes a valid topological order of $V_0, \dots, V_{k-1}$. This means that for any edges (u, v) in the original graph where $u \in V_i$ and $v \in V_j$, it must be the case that $i \leq j$. Thus it follows that *topOrder.get(n-1)* must be a vertex in V_{k-1} , and that all outgoing edges from a vertex in V_{k-1} must lead to another vertex in V_{k-1} . Thus, the first call to *dfsVisitForSCC* will visit exactly the vertices in V_{k-1} . Since all vertices in V_{k-1} have been visited, they will not be visited again, so they can be treated as no longer being in the graph. By using the same argument, it follows that the next call to *dfsVisitForSCC* will start at some vertex in V_{k-2} , and the vertices visited will be exactly those in V_{k-2} . By induction, there will be k calls to *dfsVisitForSCC* made from *stronglyConnectedComponents*, where call i for $i = k-1, k-2, \dots, 1, 0$, will create a tree in the depth-first search forest holding exactly the vertices in V_i . Furthermore, since each of the k calls creates a fresh set to hold the vertices visited during that call, components in *scc* at the end of the algorithm are exactly the sets of vertices for the k strongly connected components. ■

The *getStronglyConnectedComponents* method returns a set of sets of vertices, with one set corresponding to each strongly connected component of this graph. It throws a *GraphException* when the graph is undirected, since the strongly connected components are defined only for directed graphs.

```

public Set<Set<V>> getStronglyConnectedComponents() {
    if (!isDirected())
        throw new GraphException("applies only to directed graphs");
    sccUpdateAsNeeded();
    return scc;
}

```

Correctness Highlights: Follows from *UPDATED* and the correctness of *stronglyConnectedComponents*. ■

The *numStronglyConnectedComponents* method returns the number of strongly connected components in this graph. It throws a *GraphException* when the graph is undirected, since the strongly connected components are defined only for directed graphs.

```

public int numStronglyConnectedComponents() {
    if (!isDirected())
        throw new GraphException("applies only to directed graphs");
    sccUpdateAsNeeded();
    return scc.getSize();
}

```

53.8 Performance Analysis

We now analyze the time complexity of breadth-first search and depth-first search in terms of the graph representation methods. The key observation is that each vertex will be visited at most once in breadth-first search, and exactly once in depth-first search. It is easily seen that constant time is spent for each edge examined. Thus, the asymptotic time complexity is just the time required to iterate over the outgoing edges from each vertex in the graph. As we will show when presenting the adjacency list and adjacency matrix data structures, this leads to an $O(n^2)$ algorithm when an adjacency matrix is used as the underlying representation (Section 54.7), and an $O(n+m)$ algorithm when an adjacency list is used as the underlying representation (Section 55.7).

After running breadth-first search, the *getPathDistanceFromSource* method takes constant time, and the *getPathFromSource* method takes time proportional to the number of edges in the requested path.

After running depth-first search, the *hasCycle* method takes constant time. Similarly, the *topologicalOrder* method takes constant time since the positional collection has already been created, and a reference to it can be returned in constant time. It would take $O(n)$ time to iterate through the returned positional collection. Similarly, *getCycle* takes constant time, and it would take time proportional to the length of the cycle to examine it.

The analysis for the strongly connected components algorithm is like that for depth-first search. The overall time is the sum of the times for two calls to depth first search. Therefore, for an adjacency matrix, the strongly connected component algorithm has time complexity $O(n^2)$. For an adjacency list, the strongly connected component algorithm would run in $O(n+m)$ time.

53.9 Case Study: Garbage Collection

When an object is created, Java allocates space for it in a section of virtual memory called the **heap**. The amount of space allocated depends upon the object's internal representation, as defined by its class. (See Section A.4 of the Appendix for additional background.) In some object-oriented languages, like C++, programmers must specify when an object should be destroyed. However, Java provides **garbage collection**. That is, programmers do not specify when an object is no longer needed. Instead, Java's *garbage collector* automatically reclaims the storage occupied by objects that are no longer accessible from the executing program. Garbage collection algorithms dates back to Lisp systems developed by John McCarthy in the early 1960s. While there have been many garbage collections developed, in this section we focus McCarthy's original **mark-and-sweep algorithm**.

The runtime system maintains a **free list** of available memory objects. In Lisp, each object is a fixed size having two references (historically named *car* and *cdr*). At the end of this section, we briefly discuss how to extend this algorithm to larger objects. When the application program allocates a new object, the first object from the free list is removed and provided to the application. When the free list is empty, the garbage collection algorithm is executed.

An object is defined to be **reachable** by an executing program if there is a reference to that object on the execution stack, or if some other reachable object refers to that object. All other objects are considered to be **garbage**. The mark-and-sweep garbage collection algorithm consists of two phases:

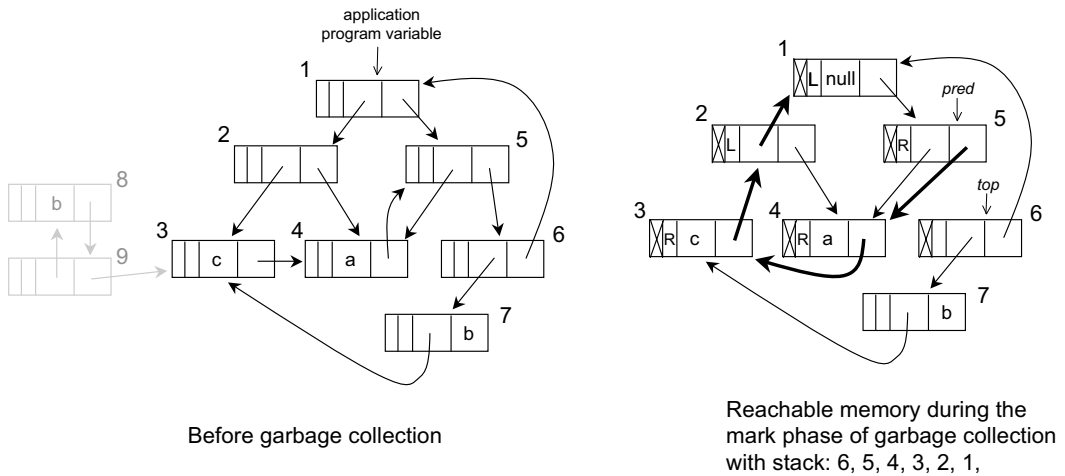


Figure 53.6

A snapshot of memory before and during in-place depth-first search for the mark phase of garbage collection. Each object is shown as a mark bit (blank for unmarked and an “X” when marked), the back bit (L or R) to indicate the reference used by the path that defines the stack (which is also the reference “borrowed”), the *left* instance variable, and the *right* instance variable. Reachable objects are shown in black and garbage is shown in gray. The graph on the right shows the reachable objects of memory during the mark phase with the borrowed references shown as a bold arrow. Note that node 7 has not yet been visited. The depth-first search stack (from the top to the bottom) is 6, 5, 4, 3, 2, 1.

Mark Phase: marks all objects that are reachable.

Sweep Phase: sweeps through all of memory moving any unmarked objects (the garbage) to the free list, and resetting any marked object to unmarked for the next execution of garbage collection.

How can the mark phase efficiently locate all reachable objects? Observe that the object heap can be modeled as an unweighted directed graph, in which each vertex is an object, and there is an edge from u to v if an instance variable from object u references object v . Both the breadth-first search (Section 53.4) and depth-first search (Section 53.5) algorithms seem to be good candidates to search this graph to find all vertices reachable from the application program's execution stack. The time complexity of both breadth-first search and depth-first search is proportional to the number of reachable (non-garbage) objects, which is clearly asymptotically optimal since each reachable object must be marked.

However, there is one major issue that must be addressed. When a garbage collection algorithm is executed, there is no free memory in the heap. In the worst case, the space requirements for the breadth-first search queue or the depth-first search stack is proportional to the number of accessible objects. An unacceptable solution to this problem would be to set aside a significant fraction of the memory for garbage collection. Instead, it is desirable to use an algorithm that can work within the confines of the existing allocated objects. Although both breadth-first and depth-first search could be used to traverse the object graph, it turns out that only depth-first search is amenable to this space-constrained garbage collection problem. Depth-first search first considers *left* and then considers *right*. If the reference being considered leads to an unmarked object, then that object is recursively visited (placed on the top of the stack). Depth-first search has the very important property that objects on the stack correspond to a path in the graph from the object at the bottom of the stack to the object at the top of the stack. In 1965, Peter Deutsch, and Herbert Schorr and W.M. Waite [133] exploited this property to design an in-place depth-first search implementation

that stores the stack using “borrowed” fields from the objects on the stack. Upon completion of the mark phase, the marked objects correspond exactly the objects that are reachable and the borrowed fields have been restored. This algorithm, known as the **Deutsch-Schorr-Waite in-place marking algorithm**, works as follows.

The key observation is that the stack that would normally exist in a recursive implementation of depth-first search can be encoded within the memory of the objects themselves. For simplicity, we assume each object has four components: the *mark* bit, the *back* bit, the *left* field, and the *right* field. The algorithm itself uses a variable *top* to reference the object at the top of the “stack,” and a variable *prev* to reference the object below it on the “stack” (or *null* if the “stack” holds only one object). Suppose that during depth-first search *top* was reached by the *left* field of *prev*. Then *prev.left* could temporarily be used to reference the object that precedes *prev* on the stack. Observe that by recording (in the *back* bit) that *prev.left* was used to reach *top*, we maintain the predecessor of the stack, and also the information needed to continue depth-first search when control is returned to *prev*. Namely if the *back* bit is “L” (for left) then the left field references the predecessor of *prev* in the stack, and the right field must still be considered. Likewise, if the back bit is “R” then the right field references the predecessor of *prev* in the stack. So after this object and all of the objects reachable from both *left* and *right* have been visited, the “stack” can be popped by letting *prev.right* = *top* and letting *top* = *prev*. This same process is used by all objects in the stack that precede *prev*. Figure 53.6 illustrates this idea by showing memory at one point during the execution of in-place depth-first search.

During depth-first search, the object at the top of the stack is being considered. First, the *left* and then the *right* fields are considered. If the field being considered references an unmarked object, then that object is placed on the top of the stack. When an object is placed on the top of the stack we say it is **visited**. When visiting object *v* from object *u*, the in-place depth-first search procedure marks *v*, updates *pred* and *top* (using a temporary variable) so that *pred* = *u* and *top* = *v*. In addition, it borrows the reference from *u* to *v* to instead reference *u*’s predecessor on the stack, and sets the back bit of *u* to indicate which reference field (*left* or *right*) was borrowed.

When all objects reachable from *v* have been visited, *v* is removed from the stack (returning control to *u*) by using the borrowed pointer of *u* (as specified by *u*’s back bit) to compute which object *x* that precedes *u* on the stack, and also to restore the borrowed reference from *u* to reference *v*. Finally, *top* and *pred* are updated so that *top* = *u* and *pred* = *x*. Then depth-first search can resume at *u*.

These ideas can be generalized to objects with *k* fields, by replacing the back bit by $\lceil \log_2 k \rceil$ bits. Thus, the total space usage for the Deutsch-Schorr-Waite algorithm is $\lceil \log_2 k \rceil + 1$ bits per object, plus three global variables (*top*, *pred*, and *temp*). With additional optimizations, the back bits (and often the mark bit) can be blended into space already allocated to each object so that the only space used are the three global variables. Aho, Hopcroft and Ullman [3] is a good source for a more details discussion of the Deutsch-Schorr-Waite algorithm, including extensions to handle variable-sized objects.

53.10 Further Reading

Ahuja, Magnanti, and Orlin [4], Even [51], and Tarjan [146] are good references on graph algorithms. Breadth-first search was independently discovered by Moore [115] and Lee [102].

Although depth-first search has been used widely since the late 1950’s, Hopcroft and Tarjan [84, 143] illustrated how useful it is for solving a variety of graph problems. The Deutsch-Schorr-Waite in-place marking algorithm was discovered independently by Peter Deutsch, and Her-

bert Schorr and W.M. Waite [133] in 1965. Further details about the Deutsch-Schorr-Waite algorithm can be found in books by Knuth [96] and Aho, Hopcroft, and Ullman [3]. Hopcroft and Tarjan [84] also recommended the use of an adjacency list over an adjacency matrix for sparse graphs. Knuth [96] developed the linear time topological sort algorithm, and Tarjan [143] gave the first linear-time algorithm for finding strongly connected components. The algorithm we provide for strongly connected components is a modification of that from Aho, Hopcroft, and Ullman [3], who in turn credit it to S.R. Kosaraju (unpublished) and M. Sharir [139].

53.11 Quick Method Reference

AbstractGraph Public Methods

- p. 858 *AbstractGraph*(GraphRepresentation<V,E> graph, boolean multigraph)
- p. 860 void **addEdge**(E e)
- p. 859 boolean **addVertex**(V v)
- p. 858 boolean **allowsMultiEdges**()
- p. 859 boolean **containsEdge**(V source, V dest)
- p. 859 boolean **containsVertex**(V v)
- p. 861 Iterator<E> **edgesFrom**(V v)
- p. 861 Iterator<E> **edgesTo**(V v)
- p. 873 Set<Set<V>> **getConnectedComponents**()
- p. 873 PositionalCollection<E> **getCycle**()
- p. 859 E **getEdge**(V source, V dest)
- p. 877 Set<Set<V>> **getStronglyConnectedComponents**()
- p. 873 boolean **hasCycle**()
- p. 858 boolean **isDirected**()
- p. 861 NonMutatingIterator<V> **iterator**()
- p. 873 int **numConnectedComponents**()
- p. 877 int **numStronglyConnectedComponents**()
- p. 859 int **numVertices**()
- p. 860 boolean **removeEdge**(E e)
- p. 860 boolean **removeVertex**(V v)
- p. 874 PositionalCollection<V> **topologicalOrder**()
- p. 866 InTree<V, E> **unweightedShortestPaths**(V s)

AbstractGraph Internal Methods

- p. 870 void **checkForCycle**(V u, E e)
- p. 872 void **dfs**()
- p. 870 void **dfsUpdateAsNeeded**()
- p. 871 void **dfsVisit**(V u, DFSData<E> uData, Set<V> component)
- p. 876 void **dfsVisitForSCC**(V u, DFSData<E> uData, Set<V> component)
- p. 861 Set<E> **edgeSet**()
- p. 859 void **graphUpdated**()
- p. 870 DFSData<E> **newDFSData**(V v, E e)
- p. 876 void **sccUpdateAsNeeded**()
- p. 876 void **stronglyConnectedComponents**()

Chapter 54

Adjacency Matrix Data Structure

package graph

GraphRepresentation<V,E extends Edge<V>> implements *Iterable*<V>

↳ AdjacencyMatrixRepresentation<V,E extends Edge<V>> implements *Iterable*<V>

Uses:

Java primitive array is used for the matrix.

DoublyLinkedList (Chapter 16) is used for the edge list for multi-edges.

Strengths: When the graph is dense, the adjacency matrix is the most space efficient representation, particularly when it is not a multigraph. If desired, the space usage can be reduced even further for an undirected graph. Another advantage of an adjacency matrix is that it takes constant time to determine if there is an edge from vertex u to vertex v . Finally, if edge (u, v) only represents the existence of a relation from u to v and there is no associated information, then an adjacency matrix can be made even more space efficient by making it a boolean matrix (or an array of bit vectors) instead of a matrix of edge references.

Weaknesses: One of the biggest drawbacks of an adjacency matrix is that it takes $O(n)$ time to iterate over all of the outgoing edges, or incoming edges, of a vertex regardless of the degree of the vertex. This is particularly problematic since for many algorithms, such as depth-first search, and breadth-first search, the dominant cost is to iterate over all of the edges in the graph. For an adjacency matrix this takes $O(n^2)$ time regardless of the number of edges, whereas for an adjacency list it takes $O(n + m)$, which is asymptotically better when $m = o(n^2)$.

Competing Data Structures: If the graph is sparse, or the application frequently needs to iterate over all outgoing, or incoming, edges of a vertex, then the adjacency list (Chapter 55) should be considered. For a sparse graph, when it is necessary to efficiently iterate over the incoming edges of a vertex, an augmented adjacency list should be considered. Finally, for a sparse graph when it is necessary to efficiently determine if there is an edge between two specified vertices, either an adjacency set or an augmented adjacency set (to support iteration over the inedges) should be considered.

54.1 Internal Representation

In this chapter we describe the adjacency matrix representation, which is implemented in the AdjacencyMatrixRepresentation class. The AdjacencyMatrix class (Section 54.6), at the end of this

chapter, is an example of an unweighted graph class that uses the adjacency matrix representation. Section 56.8 provides an example of a weighted graph class that also uses this representation.

Instance Variables and Constants: At a high-level, the adjacency matrix *edges* is an n by n matrix where *edges*[i][j] corresponds to the edge(s) from v_i (the vertex with associated id i) to v_j (the vertex with associated id j). There are two independent features that can be specified by parameters to the constructor.

Directed: For a directed graph, an edge from vertex v_i to vertex v_j , is stored in *edges*[i][j]. As discussed in the optimizations, while a more space efficient implementation is possible, in our implementation an undirected edge between v_i and v_j is stored both in *edges*[i][j] and *edges*[j][i].

Multigraph: The simplest form for an adjacency matrix is to have *edges* be a two-dimensional array of references to edges. However, this representation cannot accommodate multiple edges between the same source and destination vertices. Our implementation supports multiple edges in a space-efficient way, by having *edges*[i][j] be *null* when there is no edge from v_i to v_j , by having *edges*[i][j] references the edge from v_i to v_j when the first such edge is added, and by having *edges*[i][j] refer to a list of edges after additional edges from vertex v_i to vertex v_j have been added.

To control the space overhead of the adjacency matrix for small graphs, we define a default capacity of eight for the dimensionality of the matrix. The instance variable, *edges* is the adjacency matrix indexed by *sourceID*, *destinationID*. The type of *edges* must be *Object* since each entry may reference either an edge or an edge list.

```
public static final int DEFAULT_INITIAL_CAPACITY = 8;
Object[][] edges;                                //indexed by sourceID, destinationID
boolean isDirected;                             //true if this represents a directed graph
```

Instance Variables and Constants: Each vertex placed in the graph is assigned a unique id. The mapping from each vertex to its associated id is held in the mapping *ids*, an open addressing mapping.

```
Mapping<V, Integer> ids = new OpenAddressingMapping<V, Integer>();
```

The instance variable *nextFreeID* holds the smallest integer that has not yet been used as an id.

```
int nextFreeID = 0;
```

Populated Example: Vertex deletions affect the representation by leaving blank rows and columns in the matrix that are filled in by later vertex insertions. To illustrate how these gaps arise, Figure 54.1 shows the internal representation for the graph shown at the right side of Figure 54.2, which is obtained by deleting vertex b from the graph shown at the left side of Figure 54.2. When vertex b was removed, it was removed from *vertices*. It would take $O(n^2)$ time to reallocate a smaller matrix and copy the entries. We have instead chosen to implement a **lazy delete** in which the vertex is simply removed from the *ids* mapping. The corresponding row and column entries are set to *null*, but they remain in the matrix. If a new vertex is added later to the graph, then this space in the adjacency matrix is reused for the new vertex. However, if there is a significant decrease in the number of vertices in the graph, then the matrix would need to be resized.

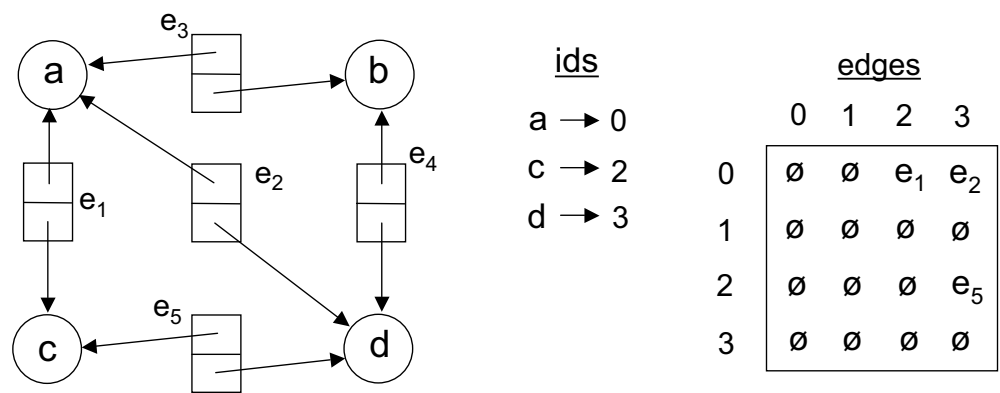


Figure 54.1
The adjacency matrix internal representation for the directed graph in the right side of Figure 54.2. Each edge instance is drawn as a rectangle split in two. The upper square references the source vertex, and the lower square references the destination vertex. For simplicity in the mapping *ids*, and the matrix *edges*, we show a reference to a vertex object or an edge object by its name, rather than showing an arrow to the instance for it.

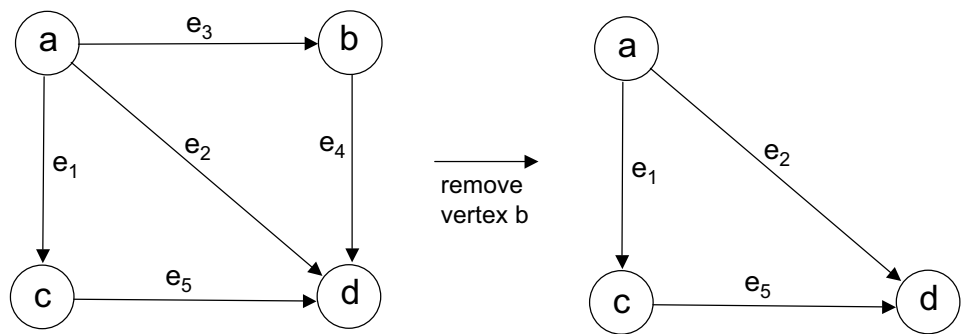


Figure 54.2
Our populated example is the graph shown on the right, which was obtained by removing vertex *b* from the graph on the left.

Terminology: Throughout this chapter we use the following terminology.

- We use v_i to denote the vertex with an associated id of i .
- For a directed graph, we say edge $e = (v_i, v_j) \in edges$ when either
 - $edges[i][j] = e$, or
 - $edges[i][j]$ is a collection of edges that contains e .
- For an undirected graph, we say edge $e = \{v_i, v_j\} \in edges$ when either
 - $edges[i][j] = e$,
 - $edges[j][i] = e$,
 - $edges[i][j]$ is a collection of edges that contains e , or
 - $edges[j][i]$ is a collection of edges that contains e .

Abstraction Function: Let AM be an adjacency matrix representation for a graph. The abstraction function is:

$$AF(AM) = (V, E) \text{ where } V = \{v \mid \exists te \in ids \text{ such that } te.tag = v\} \text{ and } E = \{e \in edges\}.$$

Design Notes: As discussed earlier, when a vertex is deleted, it would be too expensive to reallocate a new matrix and copy all n^2 entries. To maximize the space usage while minimizing the need to resize, we have introduced the id pool to reuse ids, so that empty matrix entries can be reused. It needs to be resized when the number of vertices exceeds the size of the array, and then the array is doubled (quadrupling the number of entries). The matrix's space usage is very good unless the number of vertices in the graph has been significantly reduced.

Optimizations: When the graph is undirected, the space requirement of an adjacency matrix could be reduced by a factor of two, by using only the half of the matrix above the diagonal (i.e., for $i \leq j$). We have not used this option since it would complicate the process of iterating over the outgoing or incoming edges from a vertex. If minimizing space is important, then this change could be made with slightly more code complexity in the iterator. Furthermore, when the edge object contains no information but a source and destination, each entry in the matrix could be replaced by a boolean, or the entire matrix could be replaced by an array of bit vectors.

When a significant fraction of the rows (and columns) of the matrix correspond to vertices that have been deleted, then the matrix could be compacted by reassigning the ids of the vertices in use to be $\{0, \dots, n-1\}$, allocating a new n by n matrix, and copying the matrix entries into the new matrix that can then replace *edges*. We do not include that optimization here.

54.2 Representation Properties

The following properties are maintained by the adjacency matrix representation. The first property says that vertex v is in the graph if and only if there is a tagged element in *ids* with a tag of v . **UNDIRECTEDGRAPH** is used to ensure the undirected edge $\{v_i, v_j\}$ can be represented as going from v_i to v_j or from v_j to v_i . That is, each undirected edge is treated as two directed edges, one in each direction. This allows the underlying implementation to work entirely in terms of directed edges. **DIRECTEDGRAPH** is used to ensure that the directed edge (v_i, v_j) is treated only as going from v_i to v_j . We introduce **MATRIXENTRY** to capture the use of lists in the matrix entries for handling multigraphs. The last three properties relate to the maintenance of vertex ids.

VERTICES: The keys of the *ids* mapping is exactly the set of vertices currently in the graph.

UNDIRECTEDGRAPH: For an undirected graph, the edge $\{v_i, v_j\}$ with source v_i and destination v_j , can be traversed either from v_i to v_j , or from v_j to v_i . More formally, for $i \neq j$, an edge $\{v_i, v_j\}$ is in an undirected graph if and only if $edges[i][j] \neq null$ and $edges[j][i] \neq null$.

DIRECTEDGRAPH: For a directed graph, an edge (v_i, v_j) with source v_i and destination v_j can be traversed from v_i to v_j , but not from v_j to v_i . More formally, edge (v_i, v_j) is in a directed graph if and only if $edges[i][j] \neq null$.

MATRIXENTRY : When there is at least one edge from v_i to v_j , then either $edges[i][j] = e$ where e is the only edge from v_i to v_j , or otherwise $edges[i][j]$ references a non-empty list of all edges in the graph from v_i to v_j . When there is no edge in the graph from v_i to v_j , $edges[i][j] = null$.

UNIQUEID: No two vertices in the graph have the same id.

NEXTFREEID: The instance variable *nextFreeID* is the smallest integer that has not been assigned to any vertex, including those that have been removed.

POOLOFIDS: The id pool holds every integer in $\{0, 1, \dots, nextFreeID - 1\}$ that is not assigned to a vertex currently in the graph.

For multigraphs, we use instances of the static `EdgeList` class as the lists of edges in the matrix.

```
static class EdgeList<E> extends DoublyLinkedList<E> {
    EdgeList(E x) {
        add(x);
    }
}
```

The queue *idPool* holds all integers less than *nextFreeID* that are not currently assigned to a vertex in the graph. When a vertex is removed from the graph, its id is added to the pool for later reassignment to a new vertex.

```
Queue<Integer> idPool = new Queue<Integer>();
```

The *nextID* method returns the id for a newly added vertex.

```
int nextID() {
    if (!idPool.isEmpty())
        return idPool.dequeue();
    else
        return nextFreeID++;
}
```

Correctness Highlights: By **POOLOFIDS** and the correctness of the `Queue isEmpty` method, when the id pool is not empty, the id returned is not currently in use. Also, by the correctness of `dequeue`, **POOLOFIDS** is preserved. When the id pool is empty, then by **NEXTFREEID**, the id assigned is the smallest integer not in use. Thus, **UNIQUEID** is preserved. Finally, incrementing *nextFreeID*, preserves **NEXTFREEID**. ■

54.3 Methods

In this section we describe the methods of the `AdjacencyMatrixRepresentation` class.

54.3.1 Constructors

The constructor creates a matrix dimensioned according to the default initial capacity. If desired, a constructor that accepts an initial matrix size could be added. No parameter is needed to indicate

whether the graph allows multi-edges, since the differences in the implementation for a graph and a multigraph are handled by the `GraphRepresentation` class.

```
public AdjacencyMatrixRepresentation(boolean isDirected) {
    this.isDirected = isDirected;
    edges = new Object[DEFAULT_INITIAL_CAPACITY]
        [DEFAULT_INITIAL_CAPACITY];
}
```

54.3.2 Trivial Accessors

The *isDirected* method returns *true* if and only if this graph representation is being used to support a directed graph. Typically, this choice is passed to the constructor of the graph representation.

```
public boolean isDirected() {
    return isDirected;
}
```

The *numVertices* method returns the number of vertices in this graph.

```
public int numVertices() {
    return ids.getSize();
}
```

54.3.3 Algorithmic Accessors

The *containsVertex* method returns *true* if and only if the given vertex is in this graph.

```
public boolean containsVertex(V vertex) {
    return ids.contains(vertex);
}
```

| *Correctness Highlights:* Directly follows from `MATRIXENTRY`. ■

The public *containsEdge* method takes *source*, the source vertex, and *dest*, the destination vertex. It returns *true* if and only if there is an edge in the graph from the source vertex to the destination vertex.

```
public boolean containsEdge(V source, V dest) {
    try {
        return edges[ids.get(source)][ids.get(dest)] != null;
    } catch (NoSuchElementException nsee) {
        return false;
    } catch (ArrayIndexOutOfBoundsException nsee) {
        return false;
    }
}
```

Correctness Highlights: If either vertex is not in the graph, a *NoSuchElementException* will be thrown when trying to get its id and *false* is corrected returned. Otherwise, if there is an edge from the source to the destination, then there is a non-null entry at the corresponding position in the matrix. This is the case for an undirected graph even if the edge $\{source, dest\}$ was added as $\{dest, source\}$. ■

The *getEdge* method takes *source*, the source vertex, and *dest*, the destination vertex. It returns an edge in the graph from the source vertex to the destination vertex, or *null* if there is no such edge. If the graph is undirected, the edge *e* returned may have *e.source* = *dest* and *e.dest* = *source*.

```
public E getEdge(V source, V dest) {
    try {
        Object entry = edges[ids.get(source)][ids.get(dest)];
        if (entry instanceof EdgeList)
            return ((EdgeList<E>) entry).get(0);
        else
            return (E) entry;
    } catch (NoSuchElementException nsee) {
        return null;
    } catch (ArrayIndexOutOfBoundsException nsee) {
        return null;
    }
}
```

Correctness Highlights: Directly follows from MATRIXENTRY. ■

The *iterator* method returns a non-mutating iterator over the vertices of the graph. A non-mutating iterator is returned to ensure that the application program can remove a vertex from the graph only by means of the *removeVertex* method.

```
public NonMutatingIterator<V> iterator() {
    return new NonMutatingIterator<V>(ids.tags());
}
```

54.3.4 Representation Mutators

The internal *resize* method takes *size*, the desired number of rows and columns for the matrix. This method requires that all vertex ids for vertices in the graph are at most *size* - 1. The provided implementation uses this method only to increase the size of the matrix. However, it could be used to reduce the size of the matrix if desired.

```
void resize(int size) {
    Object[][] replacement = new Object[size][size];
    for (int i = 0; i < edges.length; i++)
        System.arraycopy(edges[i], 0, replacement[i], 0, edges[i].length);
    edges = replacement;
}
```


Correctness Highlights: The system *arraycopy* is used to copy each row of the matrix into the new array with the specified size. By the requirement on this method, the abstraction function is not changed. ■

54.3.5 Content Mutators

We now describe the methods that are used to modify the structure of the graph. The *addVertex* method takes *vertex*, the vertex to add to this graph. It adds the given vertex unless it is already in the graph. It returns *true* if and only if the vertex is added to the graph.

```
public boolean addVertex(V vertex) {
    if (!ids.contains(vertex)) {
        ids.put(vertex, nextID());
        return true;
    }
    return false;
}
```

//creating mapping from vertex to id
//the vertex was created and added to the graph

//the vertex was already in the graph

Correctness Highlights: By the correctness of the mapping *contains* method, the vertex is added to the graph only when it is not already in the graph. Thus the correct value is returned. By the correctness of *nextId*, the id assigned to the new vertex preserves UNIQUEID. By the correctness of the *put* and *add* methods, the new vertex is added to the *ids* mapping and the *vertices* set. ■

The internal *addEdge* method takes *sourceID*, the id associated with the source vertex, *destID*, the id associated with the destination vertex, and *edge*, the edge to add. This method requires the ids provided for the source and destination vertices are based upon the source and destination vertices of the provided edge.

```
protected void addEdge(int sourceID, int destID, E edge) {
    if (Math.max(sourceID, destID) ≥ edges.length)
        resize(edges.length*2);
    if (edges[sourceID][destID] == null)
        edges[sourceID][destID] = edge;
    else {
        if (!(edges[sourceID][destID] instanceof EdgeList))
            edges[sourceID][destID] = new EdgeList<E>((E) edges[sourceID][destID]);
        ((EdgeList<E>) edges[sourceID][destID]).add(edge);
    }
}
```

//first edge from source to dest

//use edge list when multiple edges between sourceID and destID

Correctness Highlights: If either the *sourceID* or *destinationID* is larger than the dimension of the matrix, then *resize* is used to make the matrix large enough for the specified vertex ids. If *edges[sourceID][destID]* is *null*, then simply storing the edge in that matrix entry preserves MATRIXENTRY. Otherwise, if *edges[sourceID][destID]* does not already contain an edge list, an edge list must be allocated, and the edge currently referenced by *edges[sourceID][destID]*

must be placed in the edge list. In either case, adding the new edge to the edge list preserves MATRIXENTRY. ■

The public *addEdge* method takes *edge*, the edge to add to this graph. It requires that if the graph is not a multigraph, there must not already be an edge between the two vertices.

```

public void addEdge(E edge) {
    addVertex(edge.source());           //add source vertex if needed
    int sourceID = ids.get(edge.source()); //and get its id
    addVertex(edge.dest());             //add destination vertex if needed
    int destID = ids.get(edge.dest());    //and get its id
    addEdge(sourceID, destID, edge);     //add the edge
    if (!isDirected && sourceID != destID) //if an undirected graph and not a self loop
        addEdge(destID, sourceID, edge); //add edge in other direction
}

```

Correctness Highlights: By the correctness of *addVertex*, VERTICES and UNIQUEID are preserved. By the correctness of the internal *addEdge* method, the edge is added. Furthermore, both UNDIRECTEDGRAPH and DIRECTEDGRAPH are satisfied by adding the edge from the source vertex to destination vertex in all cases, and also by adding an edge from the destination vertex to the source vertex in an undirected graph when *edge.source* $\neq$ *edge.destination*. ■

The public *removeVertex* method takes *vertex*, the vertex to be removed, and returns *true* if and only if the vertex existed and was removed. Removing a vertex implicitly removes all of its incident edges.

```

public boolean removeVertex(V vertex) {
    if (ids.contains(vertex)) {
        int id = ids.get(vertex);           //get id of the vertex
        for (int i = 0; i < edges.length; i++) { //remove all incident edges
            edges[i][id] = null;
            edges[id][i] = null;
        }
        ids.remove(vertex);                //removing mapping entry from this vertex to its id
        idPool.enqueue(id);                //reclaim id for later use
        return true;
    }
    return false;
}

```

Correctness Highlights: By the correctness of the mapping *contains* method, if the vertex is in the graph, its associated id is returned. Setting to *null* all entries in row and column *vertexID* preserves MATRIXENTRY, since all edges that have a source id or destination id of *vertexID* are no longer in the graph. The vertex is correctly removed from the mapping from the vertices to the ids and its id is returned to the pool. Therefore, VERTICES, UNIQUEID, NEXTFREEID, and POOLOFIDS are preserved. ■

The internal *removeEdge* method takes *sourceID*, the id assigned to the source vertex, and *destID*, the id assigned to the destination vertex. It returns *true* when the edge is successfully removed, and *false*, if the edge did not exist. It requires that *edge.source* has id *sourceID* and *edge.dest* has id *destID*.

```

boolean removeEdge(int sourceID, int destID, E edge) {
    Object entry = edges[sourceID][destID];
    if (entry == null)                                     //no such edge to remove
        return false;
    if (entry == edge) {                                     //single edge from sourceID to destID
        edges[sourceID][destID] = null;
        return true;
    } else if (entry instanceof EdgeList) {                //list of >= 1 edge from sourceID to destID
        ((EdgeList) entry).remove(edge);                  //remove an edge from the list
        if (((EdgeList) entry).isEmpty())                  //preserve MatrixEntry
            edges[sourceID][destID] = null;
        return true;
    }
    return false;
}

```

Correctness Highlights: By MATRIXENTRY when `edges[sourceID][destID]` is `null`, `false` is properly returned. Likewise, by MATRIXENTRY, when `edges[sourceID][destID]` references an edge, it is the only edge in the graph from the source to destination. Setting this entry to `null` both removes the edge and preserves MATRIXENTRY. The remaining possibility is that `edges[sourceID][destID]` references a list of at least one edge from the source to destination. By the correctness of the list `remove` method, one of these edges is removed. Finally, if the edge list is now empty, then `edges[sourceID][destID]` is set to `null` to preserve MATRIXENTRY. Since `false` is returned when the edge requested is not in the graph, `true` is returned exactly when the edge is removed. ■

The public `removeEdge` method takes `edge`, the edge to be removed from this graph. It returns `true` when the edge is successfully removed, and `false`, if the edge did not exist.

```

public boolean removeEdge(E edge) {
    if (!ids.contains(edge.source()) || !ids.contains(edge.dest()))
        return false;
    int sourceID = ids.get(edge.source());
    int destID = ids.get(edge.dest());
    if (removeEdge(sourceID, destID, edge)) {                //returns true if edge was in graph
        if (!isDirected && destID != sourceID)
            removeEdge(destID, sourceID, edge);
        return true;
    }
    return false;
}

```

Correctness Highlights: By the correctness of the internal `removeEdge` method, the edge is removed if it exists in the graph. Furthermore, if the graph is directed and the edge is not a self loop, removing the edge from the destination vertex to the source vertex preserves UNDIRECTED-GRAPH. ■

54.4 Edge Iterators

The *edgesFrom* method takes *source*, the source vertex. It returns an iterator over the outgoing edges from *source*. Observe that for an undirected graph, the iterator is over all edges incident to *source*. It throws an *IllegalArgumentException* when *source* is not in the graph.

```
public Iterator<E> edgesFrom(V source) {
    if (!ids.contains(source))
        throw new IllegalArgumentException(source + " is not in the graph.");
    return (Iterator<E>) new IncidentEdgeIterator(ids.get(source), true);
}
```

Correctness Highlights: Follows from the correctness of the incident edge iterator that is described in Section 54.5. ■

The *edgesTo* method takes *dest*, the destination vertex. It returns an iterator over the incoming edges to *dest*. It throws an *IllegalArgumentException* when *dest* is not in the graph.

```
public Iterator<E> edgesTo(V dest) {
    if (!ids.contains(dest))
        throw new IllegalArgumentException(dest + " is not in the graph.");
    return (Iterator<E>) new IncidentEdgeIterator(ids.get(dest), false);
}
```

Correctness Highlights: Follows from the VERTICES and the correctness of the incident edge iterator. ■

54.5 Incident Edge Iterator Inner Class

IncidentEdgeIterator implements *Iterator<E>*

To iterate over all outgoing edges from v_i , it is necessary to iterate over all entries of row i of the matrix, including iterating over any edge lists referenced from row i . To iterate over all of the incoming edges for v_i , it is necessary to iterate over column i instead of row i .

The current position of the incident edge iterator is stored as row and column indices of the matrix. We use the convention of conceptually treating row -1 as a row above the top row of the matrix, and column -1 as a column to the left of the first column. Also, for a multigraph, the iterator *subIterator* is an iterator at the current position in the edge list *entries[row][column]*.

To support iteration over either row i or column i , we specify a row increment (*rowInc*), and a column increment (*colInc*) for the incident edge iterator. To iterate over all matrix entries in a row, the row increment *rowInc* is 0 and the column increment is *colInc* is +1, and to iterate over all matrix entries in a column, *rowInc* is +1 and *colInc* is 0.

```
int row, col; //current matrix entry (row, col)
Iterator subIterator; //iterator within list at (row, col)
```

```

int rowInc;                                //0 for outedges, +1 for inedges
int colInc;                                //0 for inedges, +1 for outedges
E lastEdgeReturned;                       //the most recent return value from next

```

The incident edge iterator constructor takes *vertexID*, the id for the desired vertex, and *outgoing*, which is *true* when the desired action is to iterate over the outgoing edges from the given vertex, and *false* when the desired action is to iterator over the incoming edges from the given vertex.

```

public IncidentEdgeIterator(int vertexID, boolean outgoing) {
    if (outgoing) {
        row = vertexID;                //iterate over row vertexID
        col = -1;                      //begin just "left" of first entry
        rowInc = 0; colInc = 1;        //add one to col to move forward
    } else {
        col = vertexID;                //iterate over column vertexID
        row = -1;                      //begin just "above" the first entry
        rowInc = 1; colInc = 0;        //add one to the row to move down
    }
}

```

Correctness Highlights: We argue the constructor places the iterator in a position just before the first matrix entry to be examined, and that incrementing the current row index by *rowInc* and the current column index by *colInc* will move to the next matrix entry according to *outgoing*.

Outgoing is true: The specified behavior is to iterate over the matrix entries in row *vertexID*. Thus *row* should never change, which is satisfied by having *rowInc* set to 0. Moving to the next entry in the row corresponds to incrementing the current column index by 1, which is the behavior that will occur by setting *colInc* to 1. Since the first possible edge is in *edges[vertexID][0]*, setting *col* to -1 initializes the iterator so that advancing once, which increments *col* by 1, places the iterator at the *edges[vertexID][0]*.

Outgoing is false: Analogous to the other case, except for the goal is to iterate over the matrix entries in column *vertexID*. To iterate over a column, *col* should not change, so *colInc* should be 0, and moving forward is achieved by incrementing *row* by 1, which is the behavior that will occur with *rowInc* = 1. Finally, setting *col* = *vertexID* and *row* = -1 initializes the iterator so that advancing to the next position will place the iterator at *edges[0][vertexID]*. ■

The *hasNext* method returns *true* if and only if there is another edge in the row, or column, being considered. To improve efficiency, the iterator is moved forward so that it is placed just before the next edge to prevent the repetition of computation when *hasNext* is followed by *next* as is typically the case.

```

public boolean hasNext() {
    if (subIterator != null) {          //Case 1: when iterating through a list in edges[row][col]
        if (subIterator.hasNext())      //Case 1a: more in current edge list
            return true;
        else                            //Case 1b: if no more entries, prevent examining again
            subIterator = null;
    }
    while (row+rowInc < edges.length
           && col+colInc < edges.length) { //Case 2: find next non-null entry

```

```

    if (edges[row+rowInc][col+colInc] == null) {           //when next matrix entry is null
        row += rowInc; col += colInc;                      //move forward
    } else {                                               //Case 2a: found a non-null entry that could be a list or an edge
        if (edges[row+rowInc][col+colInc] instanceof EdgeList) {
            row += rowInc; col += colInc; //move to entry with edge list
            subIterator = ((Iterable) edges[row][col]).iterator(); //init list iterator
        }
        return true;
    }
}
return false;                                           //Case 2b: all remaining entries in desired row/col are null
}

```

Correctness Highlights: In the correctness argument for the constructor, we argued that adding *rowInc* to *row*, and *colInc* to *col*, moves the locator to the next matrix entry in the desired row (for *outgoing* = *true*) and column (for *outgoing* = *false*). We consider the following cases:

Case 1: The sub-iterator is non-null. We further divide this case into two cases based on whether or not the sub-iterator has a next item.

Case 1a: The sub-iterator has a next item. By the correctness of the sub-iterator *hasNext* function, there is another edge when it returns *true*.

Case 1b: The sub-iterator has no next item. By the correctness of the sub-iterator *hasNext* function, when it returns *false* there are no other edges in the current list being considered. To avoid reexamining the list, *subIterator* is set to *null*. This case then proceeds to Case 2 since it can be treated as if *subIterator* was *null* at the start.

Case 2: The sub-iterator is null. By MATRIXENTRY, there is another edge in the iteration order, exactly when some non-*null* matrix entry is found in the desired row (or column). As long as the next matrix entry in the iteration order is *null*, the iterator is moved to the next entry. We now consider the following subcases.

Case 2a: *edges[row + rowInc][col + colInc]* is not null. By MATRIXENTRY, there must be at least one edge from vertex *row + rowInc* to vertex *col + colInc*. Thus, *true* is the correct return value. When *edges[row + rowInc][col + colInc]* references an edge list, then to improve the efficiency of *next*, the iterator is moved to this matrix entry, and *subIterator* is initialized to be at FORE (just before the first entry) of this list.

Case 2b: All remaining matrix entries in the iteration are null. By MATRIXENTRY, *false* is properly returned. ■

The *next* method returns a reference to the next edge in the iteration order. It throws a *NoSuchElementException* when there is no next edge.

```

public E next() {
    if (!hasNext())
        throw new NoSuchElementException();
    if (subIterator != null)
        return lastEdgeReturned = (E) subIterator.next(); //move forward in sub-iterator
    else {
        row += rowInc; col += colInc;                      //move to the next matrix entry
    }
}

```

```

        return lastEdgeReturned = (E) edges[row][col];
    }
}

```

Correctness Highlights: By the correctness of *hasNext*, the exception is properly thrown if there is no next item, and *hasNext* leaves the iterator positioned so that either (1) *subIterator* is not *null*, and the sub-iterator is positioned in the edge list just before the next edge in the iteration order, or (2) the next matrix entry refers to the next edge in the iteration order. Thus, the correct value is returned. ■

The *remove* method removes the edge most recently returned by this iterator. It throws a *NoSuchElementException* when *next* has not been called since the most recent call to *remove*.

```

public void remove() {
    if (lastEdgeReturned == null)
        throw new NoSuchElementException();
    removeEdge(lastEdgeReturned);
    lastEdgeReturned = null;
}
}

```

54.6 Adjacency Matrix Class

package graph

AbstractGraph<V, E extends Edge<V>> implements Graph<V,E>

└─ AdjacencyMatrix<V,E extends Edge<V>> implements Graph<V,E>

The AdjacencyMatrix class just contains two constructors, that create a graph using an adjacency matrix representation. The caller can specify whether or not the graph is directed, and whether the graph is a multigraph. With an adjacency matrix, support is always provided to iterate through the incoming edges for each vertex.

```

public AdjacencyMatrix(boolean directed, boolean allowsMultiEdges) {
    super(new AdjacencyMatrixRepresentation<V, E>(directed), allowsMultiEdges);
}

```

The constructor with no arguments creates a directed graph that is not a multigraph.

```

public AdjacencyMatrix() {
    this(true, true);
}

```

method	time complexity (not a multigraph)	time complexity (multigraph)
<i>addEdge(e)</i>	$O(1)$	$O(1)$
<i>addVertex(v)</i>	$O(1)$	$O(1)$
<i>containsEdge(v_i, v_j)</i>	$O(1)$	$O(1)$
<i>containsVertex(v)</i>	$O(1)$	$O(1)$
<i>edgesFrom(v)</i>	$O(1)$	$O(1)$
<i>edgesTo(v)</i>	$O(1)$	$O(1)$
<i>isDirected(), isMultiGraph()</i>	$O(1)$	$O(1)$
<i>iterator()</i>	$O(1)$	$O(1)$
<i>numVertices()</i>	$O(1)$	$O(1)$
<i>getEdge(v_i, v_j)</i>	$O(1)$	$O(c_{ij})$
<i>removeEdge(e)</i>	$O(1)$	$O(c_{ij})$
<i>removeVertex(v)</i>	$O(n)$	$O(n)$
<i>addEdge(e)</i> , when resizes <i>edgeSet()</i>	$O(n^2)$ $O(n^2)$	$O(n^2)$ $O(n^2 + m)$

Table 54.3 Summary of the asymptotic time complexities for maintaining a graph using an adjacency matrix. We use c_{ij} to denote the number of edges in a multigraph from v_i to v_j . When the graph is not a multigraph, $c_{ij} = 1$.

54.7 Performance Analysis

The asymptotic time complexities of the graph representation methods for an adjacency matrix are given in Table 54.3, and the asymptotic time complexities for the edge iterator methods are given in Table 54.4.

The *isDirected*, *allowsMultiEdges*, and *numVertices* methods just return the value of an instance variable, so they take constant time. Likewise, *iterator* takes constant time since it just initializes an iterator over the set *vertices*.

There is constant expected cost to determine if a vertex is in *vertices*, and to get the id for a vertex if it exists. Thus, the *containsVertex* method takes constant time. The *edgesFrom* and *edgesTo* methods take expected constant time to obtain the id for the specified vertex, and then constant time to initialize the incident edge iterator.

Next we consider the cost to add or remove a vertex from the graph. It just takes constant time to assign the vertex an id from the id pool and expected constant time to add the vertex to *ids*. If the id for the vertex corresponds to a row/column of the matrix that exists, then no other computation is needed. However, if the vertex id assigned to the new vertex does not correspond to a row/column already in the matrix (i.e., $id \geq \text{edges.length}$) then the first edge added that involves this vertex will cause the matrix to resize. In the table, we show this cost under “*addEdge(e)*, resized.”

For all remaining methods, the *ids* mapping can find the id associated with a given vertex in expected constant time. We focus on the computation that must be performed after the id has been obtained. To remove a vertex from the graph, the $2n - 1$ entries in the row and column whose indices are the id of the removed vertex must be set to *null*. Finally the id is returned to the id pool, which takes constant time. Thus the overall cost to remove a vertex from the graph is linear in the number of vertices.

We now consider the cost of adding an edge. After obtaining the ids for the two endpoints, it takes constant time to access *edges*[*i*][*j*]. If the graph is not a multigraph, then the edge can be stored in

edge iterator method	average time complexity
constructor	$O(1)$
<i>advance()</i> when preceded by <i>hasNext</i>	$O(1)$
<i>next()</i> when preceded by <i>hasNext</i>	$O(1)$
<i>remove()</i> , not a multigraph	$O(1)$
<i>remove()</i> , multigraph	$O(c_{ij})$
<i>advance()</i>	$O(n/d)$
<i>hasNext()</i>	$O(n/d)$
<i>next()</i>	$O(n/d)$

Table 54.4 Summary of the time complexities for the incident edge iterator methods for an adjacency matrix. We use d to denote the number of edges in the iteration sequence for this iterator. For example, if the iterator is created by *edgesFrom*(v) then d is the outdegree of v . Likewise, if the iterator is created by *edgesTo*(v) then d is the indegree of v .

that entry in constant time. When the graph is a multigraph, it also takes constant time since both creating an edge list and adding to it takes constant time. For *containsEdge*(v_i, v_j), first the ids are retrieved. By *MATRIXENTRY*, if there is an edge from v_i to v_j , it must be stored in *edges*[i][j]. Thus, it is sufficient to check if *edges*[i][j] is *null*, which takes constant time. For a graph that is not a multigraph, *getEdge*(v_i, v_j) can return the edge (if any) stored in *edges*[i][j] which takes constant time. However, for a multigraph, *getEdge*(v_i, v_j) may need to iterate through the edge list referenced by *edges*[i][j]. Thus, for a multigraph *getEdge* takes $O(c_{ij})$ time where c_{ij} is the number of edges from v_i to v_j . Once an edge has been located, it can be removed in constant time. Thus *removeEdge*, has the same time complexity as *getEdge*.

The *edgeSet* method iterates through the entire matrix. When the graph is not a multigraph, this has $O(n^2)$ cost. For a multigraph, *edgeSet* must also iterate through each matrix entry and each edge list, yielding an asymptotic time complexity of $O(n^2 + m)$.

We now analyze the time complexity of the incident edge iterator methods. The constructor clearly takes constant time. Observe that for all other methods, at most n matrix entries are examined, and by definition there are d edges that occur in the iteration sequence. Thus the average cost to advance to the next edge is $O(n/d)$. Since *hasNext* advances the iterator to just before the next edge, if *advance* or *next* is called after *hasNext*, it has worst-case constant cost.

54.8 Quick Method Reference

AdjacencyMatrixRepresentation Public Methods

- p. 888 *AdjacencyMatrixRepresentation*(boolean isDirected)
- p. 891 void **addEdge**(E edge)
- p. 890 boolean **addVertex**(V vertex)
- p. 888 boolean **containsEdge**(V source, V dest)
- p. 888 boolean **containsVertex**(V vertex)
- p. 893 Iterator<E> **edgesFrom**(V source)
- p. 893 Iterator<E> **edgesTo**(V dest)
- p. 889 E **getEdge**(V source, V dest)
- p. 888 boolean **isDirected**()

- p. 889** `NonMutatingIterator<V> iterator()`
- p. 888** `int numVertices()`
- p. 892** `boolean removeEdge(E edge)`
- p. 891** `boolean removeVertex(V vertex)`

AdjacencyMatrixRepresentation Internal Methods

- p. 890** `void addEdge(int sourceID, int destID, E edge)`
- p. 887** `int nextID()`
- p. 891** `boolean removeEdge(int sourceID, int destID, E edge)`
- p. 889** `void resize(int size)`

Chapter 55

Adjacency List Data Structure

package graph

GraphRepresentation<V, E extends Edge<V>> implements *Iterable*<V>

└─ AdjacencyListRepresentation<V, E extends Edge<V>> implements *Iterable*<V>

Uses: The graph algorithm implementations use many of the other data structures presented in this book, most notably:

CircularArray (Chapter 12) is used to hold the topological order of an acyclic directed graph, and for a directed cycle in a cyclic directed graph.

Queue (Chapter 18) is used for the id pool and for breadth-first search.

OpenAddressing (Chapter 22) is used for the set of vertices.

OpenAddressingMapping (Section 49.7.2) is used for the mapping from each vertex to its id, and for the mapping from each vertex to its associated data.

TaggedBucketCollection (Chapters 50) is used for the mapping from each vertex to the set of outgoing edges, and the set of incoming edges.

Strengths: For a sparse graph, an adjacency list is very space efficient, especially if the application does not need to iterate over the incoming edges of a directed graph. Many graph algorithms iterate over the vertices in some data-dependent order, and for each vertex iterate over its outgoing edges. An adjacency list supports this operation very efficiently, allowing it to iterate over all edges in $O(m + n)$ times as opposed to $O(n^2)$ time for an adjacency matrix. For a graph with significantly fewer than n^2 edges, this reduction in time complexity can be significant.

Weaknesses: For a dense graph, an adjacency list uses more space than an adjacency matrix. Furthermore, in order to support efficient iteration over the incoming edges of a vertex in a directed graph, it is necessary to keep separate lists of the incoming and outgoing edges of each vertex, which roughly doubles the space usage. When the outgoing edges for each vertex are stored as a list, it takes time proportional to the degree of vertex u to determine if there is an edge from u to v . However, this cost can be reduced to expected constant time by replacing the lists of outgoing (and incoming) edges by sets (Chapter 20), where the hasher depends only on the source and destination vertices.

Competing Data Structures: An adjacency matrix (Chapter 54) should be considered if the graph is dense (roughly $n^2/4$ edges or more), or if the application needs to determine very efficiently whether or not there is an edge from vertex u to vertex v .

55.1 Internal Representation

In this chapter we describe the adjacency list graph representation, which is implemented in the `AdjacencyListRepresentation` class. The `AdjacencyList` class at the end of this chapter provides an example of an unweighted graph class that uses the adjacency list representation. Section 56.9 provides an example of a weighted graph class that also uses this representation.

Instance Variables and Constants: The central data structure is a mapping from each vertex to a list of the outgoing edges from that vertex. There are three independent features that can be varied, as specified by the user program through the constructor parameters.

Directed: An adjacency list most naturally represents a directed graph, since each vertex is associated with a set of outgoing edges. However, an undirected graph can be represented by storing edge $\{u, v\}$ in both the outgoing edge list for u and the outgoing edge list for v . The edge $\{u, u\}$ need not be store twice in the same list. Note that a second edge object is not created, so the outgoing edge list for v will reference an edge with a source vertex of u and a destination vertex of v . However, when $\{u, v\}$ is encountered in the outgoing edges from v , it must be treated as if v is the source and u is the destination. One disadvantage of an adjacency list is the extra space required to include the edge in both lists when the graph is undirected.

Multigraph: An adjacency list is very well suited to supporting multi-edges. The outgoing list of edges can include any number of edges, without any change required if there are multiple edges going to the same vertex.

Store Incoming Edges: One significant weakness of an adjacency list is that it does not support efficient iteration over the incoming edges of a vertex when the graph is directed. (Since undirected edges are referenced in the outgoing edges from both its source and destination, all edges incident to a vertex in an undirected graph are easily accessed.) To support efficient iteration over the incoming edges in a directed graph, an application program can specify that a list of incoming edges be maintained for each vertex. However, exercising this option approximately doubles the space usage and adds small constant amount of overhead for adding and removing each edge. Throughout this chapter we use the phrase **adjacency list of vertex v** to refer to both the list of outgoing edges and the list of incoming edges (if maintained) of vertex v .

We maintain a set *vertices* of all the vertices in the graph. Since the removal of a vertex is relatively rare, we use open addressing since it is more space efficient than separate chaining. We use a bucket mapping (Chapter 50) from each vertex to a list of its outgoing edges. Likewise, when the list of incoming edges is to be maintained for each vertex, we use a bucket mapping from the vertices to a list of incoming edges. The boolean *isDirected* indicates whether or not the graph is a directed graph.

```
Set<V> vertices = new OpenAddressing<V>();           //set of vertices in graph
TaggedBucketCollection<V, E> outEdges;             //maps each vertex to a bucket of outgoing edges
TaggedBucketCollection<V, E> inEdges;              //maps each vertex to a bucket of incoming edges
boolean isDirected = true;                          //by default, the graph is directed
```

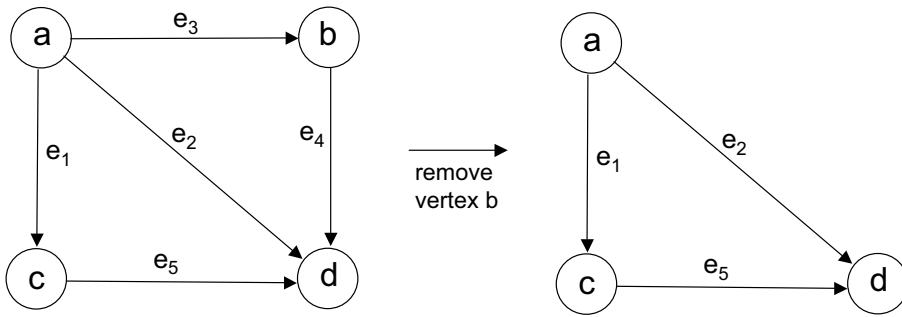


Figure 55.1

The graph on the right shows the result of removing vertex *b* from the graph on the left.

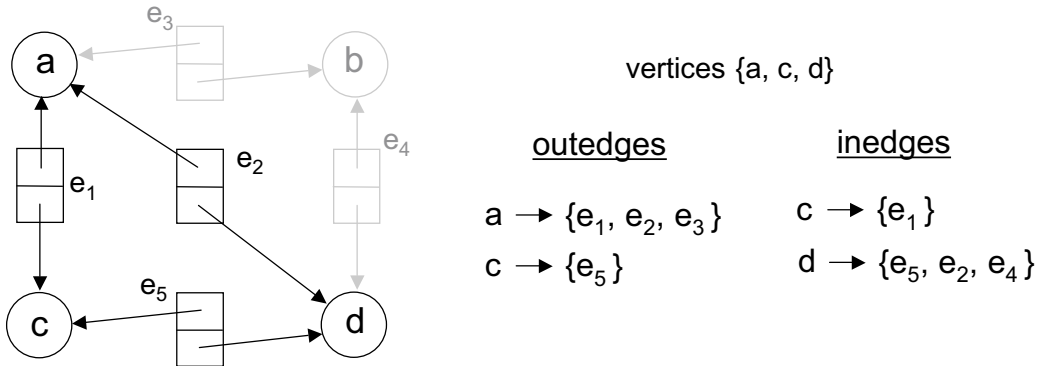


Figure 55.2

The adjacency list internal representation for the graph on the right in Figure 55.1. Each edge instance is drawn as a rectangle split in two. The upper half refers to the source vertex, and the lower half refers to the destination vertex. For simplicity in *vertices*, *outEdges*, and *inEdges*, we represent a reference to a vertex object or an edge object by name, rather than by showing an arrow to the instance for it. The vertices and edges that are no longer in use are shown in gray.

Populated Example: Vertex deletions affect the representation by leaving incident edges in the adjacency lists of the other incident vertices. To illustrate how deletions affect the representation, we consider an example graph from which a vertex has been deleted. The graph shown in the right of Figure 55.1 was obtained by deleting vertex *b* from the graph shown at the left. The adjacency list internal representation for the graph following deletion of *b* is shown in Figure 55.2. When vertex *b* is removed, it is removed from *vertices*, and also it is removed from *outEdges* and *inEdges*. However, for edge $e_3 = (a, b)$, a reference remains in the set of outedges for *a*. The first time it is encountered when iterating over the outedges of *a*, it is removed. Similarly e_4 remains in the list of inedges for *d*.

Abstraction Function: Let *AL* be an adjacency list representation. For a vertex $v \in \text{vertices}$, let v_{out} be the bucket mapped from *v* in *outedges*, and let v_{in} be the bucket mapped from *v* in *inedges*. The abstraction function for a directed graph is:

$$AF(AL) = (V, E) \text{ where } V = \{v \in \text{vertices}\} \text{ and } E = \{e = (u, v) \mid e \in u_{out} \wedge (u \in V \wedge v \in V)\}.$$

The abstraction function for an undirected graph differs in that:

$$E = \{e = \{u, v\} \mid (e \in u_{out} \vee e \in v_{out}) \wedge (u \in V \wedge v \in V)\}.$$

Data Structure	<i>storeIncomingEdges</i>	TaggedBucketCollection type
AdjacencyList	<i>false</i>	$V \rightarrow \text{List}\langle E \rangle$
AugmentedAdjacencyList	<i>true</i>	$V \rightarrow \text{List}\langle E \rangle$
Adjacency Set (no multi-edges)	<i>false</i>	$V \rightarrow \text{Set}\langle E \rangle$
Augmented Adjacency Set (no multi-edges)	<i>true</i>	$V \rightarrow \text{Set}\langle E \rangle$
Adjacency Set (with multi-edges)	<i>false</i>	$V \rightarrow \text{BucketMapping} \langle V, \text{List} \langle E \rangle \rangle$
Augmented Adjacency Set (with multi-edges)	<i>true</i>	$V \rightarrow \text{BucketMapping} \langle V, \text{List} \langle E \rangle \rangle$

Table 55.3 An overview of the adjacency list variations.

That is, an undirected graph is represented by treating edge $\{u, v\}$ as if it were the two directed edges (u, v) and (v, u) . The edge object is not duplicated but rather it is referenced by the outedges list for u and v .

Design Notes: The set of vertices is kept separately from the keys of the bucket mapping because some vertices may not have any edges and would therefore not appear as keys in the mapping. When a vertex is deleted from the graph, its entire lists of outgoing and incoming edges are removed. However, each of those edges still appears in the adjacency lists of the other incident vertices. It would be inefficient to traverse all of those lists to find and remove those “dangling” edges. Therefore, we instead have chosen to remove them lazily. More specifically, we leave the dangling edges in the list and delete them the first time they are encountered by an iterator. Detection of such an edge is made on the basis of whether or not its endpoints remain in the set of vertices.

Optimizations: In the provided implementation, edges are stored in doubly-linked lists. When the number of outgoing edges from a vertex is small, or the algorithm will iterate over all the edges, then this is a good approach. However, with this implementation the cost of locating an edge from vertex u to vertex v is proportional to the number of outgoing edges that vertex u has. A very easy change that would reduce the cost of locating an edge to expected constant time would be to replace the doubly-linked list by a set*. We call this variation an **adjacency set**. When edges are not removed frequently, then open addressing would be a good choice. Otherwise, separate chaining could be used. Using a set, instead of a doubly-linked list, would add a small amount of overhead in computing the hash function. However, if the hash table is appropriately sized, then there is no increase in the space usage.

*Since a set does not allow duplicates, when multi-edges are allowed, then the doubly-linked list must be replaced by a BucketMapping in which the destination vertex is the key and a list of edges is the associated data.

When the set of incoming edges for each vertex is maintained, another optimization would be to replace the use of the doubly-linked lists for maintaining the outedges and inedges by a single cross-linked data structure. In particular, each list item would have four references: incoming next, incoming previous, outgoing next, and outgoing previous. Each list item would appear in the outgoing and incoming edge lists for the source and destination vertices, respectively. Along with using less space than two separate lists, when vertex v is removed, all edges incident to v can be found and removed from their respective lists in time proportional to the degree of v . Although this optimization would require a customized list implementation, it would save time during iteration and simplify the iterator implementation, since the iterator would not have to be concerned with whether or not an edge is incident to a deleted vertex.

Variations: The provided implementation is for an adjacency list (when *storeIncomingEdges* is *false*) and an augmented adjacency list (when *storeIncomingEdges* is *true*). Table 55.1 summarizes other variations that could be created easily based on the value for *storeIncomingEdges* provided to the constructor, and the type of the bucket used for the tagged bucket collections *outEdges* and *inEdges*. Some methods, such as *containsEdge*, would need some minor modifications to reflect the bucket type selected.

55.2 Representation Properties

We introduce the following properties that are maintained by the adjacency list representation class. The first property says that *vertices* maintains the vertices in the graph. **DIRECTEDGRAPH** specifies that each directed edge (u, v) must be in the outedges of u . **UNDIRECTEDGRAPH** specifies that for an undirected graph, the edge $\{u, v\}$ with source u and destination v , can be traversed either from u to v , or from v to u . That is, for $u \neq v$, the edge $\{u, v\}$ is in an undirected graph if and only if both the directed edges (u, v) and (v, u) are represented. Finally, **INEDGES** ensures that for an augmented adjacency list, the directed edge (u, v) is in the outedges for u if and only if it is in the inedges for v . Recall that the inedges are never maintained for an undirected graph.

VERTICES: The set *vertices* includes exactly those vertices that are currently in the graph.

DIRECTEDGRAPH: A directed graph contains edge $e = (u, v)$ if and only if $u \in \text{vertices}$, $v \in \text{vertices}$, and *outEdges.get(u)* contains e .

UNDIRECTEDGRAPH: An undirected graph contains edge $e = \{u, v\}$ if and only if

- $u \in \text{vertices}$,
- $v \in \text{vertices}$,
- *outEdges.get(u)* contains (u, v) , and
- *outEdges.get(v)* contains (u, v) .

INEDGES: For an undirected graph, or a directed graph for which the argument to the constructor indicated that the incoming edges should not be maintained, *inEdges* remains *null*. If *inEdges* is not *null* and $e = (u, v)$ with $u, v \in \text{vertices}$, then $e \in u.\text{outedges}$ if and only if $e \in v.\text{inedges}$.

55.3 Methods

We now describe the `AdjacencyListRepresentation` methods for adding, removing, and accessing the vertices and edges of the graph.

55.3.1 Constructors

The constructor takes *isDirected*, which is true when the graph is a directed graph, and *storeIncomingEdges*, which is *true* when the incoming edges should be maintained. Observe that no parameter is needed to indicate whether the graph allows multi-edges, since the underlying representation of an adjacency list is not affected by this choice. The differences in the implementation for a graph and a multigraph are handled by the `GraphRepresentation` class.

```
public AdjacencyListRepresentation(boolean isDirected, boolean storeIncomingEdges) {
    this.isDirected = isDirected;
    outEdges = new TaggedBucketCollectionWrapper<V, E>
        (new OpenAddressingMapping<V, Collection<E>>(), DoublyLinkedList.class);
    if (isDirected && storeIncomingEdges)
        inEdges = new TaggedBucketCollectionWrapper<V, E>
            (new OpenAddressingMapping<V, Collection<E>>(), DoublyLinkedList.class);
}
```

Correctness Highlights: `INEDGES` is preserved, by initializing *inEdges* to an empty bucket collection when the graph is directed and the incoming edges are to be maintained. Otherwise, *inEdges* will remain *null*, since this is the only place in which a bucket collection is allocated. ■

55.3.2 Trivial Accessors

The *isDirected* method returns *true* if and only if this graph representation is being used to support a directed graph.

```
public boolean isDirected() {
    return isDirected;
}
```

The *numVertices* method returns the number of vertices in this graph.

```
public int numVertices() {
    return vertices.getSize();
}
```

55.3.3 Algorithmic Accessors

The *containsVertex* method returns *true* if and only if the given vertex is in this graph.


```

public boolean containsVertex(V vertex) {
    return vertices.contains(vertex);
}

```

The *containsEdge* method takes *source*, the source vertex, and *dest*, the destination vertex. It returns *true* if and only if there is an edge in the graph from the source vertex to the destination vertex.

```

public boolean containsEdge(V source, V dest) {
    if (outEdges.contains(source)) {
        Iterator<E> it = edgesFrom(source);
        while (it.hasNext()) {
            E nextEdge = it.next();
            if ((nextEdge.dest() == dest && nextEdge.source() == source) ||
                (!isDirected && nextEdge.dest() == source && nextEdge.source() == dest))
                return true;
        }
    }
    return false;
}

```

Correctness Highlights: By the correctness of the edge iterator, both incident vertices of *nextEdge* are in *vertices*. There are two cases, depending on whether or not the graph is directed. If the graph is directed then by DIRECTEDGRAPH, there is an edge *e* from *source* to *dest* if and only if *e* appears in the outedges of *source*. Since all outedges of *source* are considered, the correctness follows for this case.

Next we consider when the graph is undirected. By UNDIRECTEDGRAPH, an undirected edge *e* between *source* and *dest* must occur in the outedges of *source* as either the directed edge (*source*, *dest*) or (*dest*, *source*). Both of these cases are considered when the graph is undirected, so the correctness follows. ■

The *getEdge* method takes *source*, the source vertex, and *dest*, the destination vertex. It returns an edge in the graph from the source vertex to the destination vertex, or *null* if no such edge exists.

```

public E getEdge(V source, V dest) {
    if (outEdges.contains(source)) {
        Iterator<E> it = edgesFrom(source);
        while (it.hasNext()) {
            E nextEdge = it.next();
            if ((nextEdge.dest() == dest && nextEdge.source() == source) ||
                (!isDirected && nextEdge.dest() == source && nextEdge.source() == dest))
                return nextEdge;
        }
    }
    return null;
}

```

Correctness Highlights: The correctness is like that for *containsEdge*, except that the edge, when found, is returned. When the edge is not found, *null* is returned. ■

The *iterator* method returns an iterator over the vertices of the graph. A non-mutating iterator is returned to ensure that the application program can remove a vertex from a graph only by means of the *removeVertex* method.

```
public NonMutatingIterator<V> iterator() {
    return new NonMutatingIterator<V>(vertices.iterator());
}
```

55.3.4 Content Mutators

We now describe methods that modify the structure of the graph. The *addVertex* method takes *vertex*, the vertex to add to this graph. It adds the given vertex unless it is already in the graph. It returns *true* if and only if the vertex is added to the graph.

```
public boolean addVertex(V vertex) {
    if (!vertices.contains(vertex)) {
        vertices.add(vertex);           //add vertex to set of vertices
        return true;
    }
    return false;                      //the vertex was already in the graph
}
```

The internal *addEdge* method takes *vertex*, the vertex to be treated as the source vertex, and *edge*, the edge to add.

```
void addEdge(V vertex, E edge) {
    outEdges.put(vertex, edge);
    if (inEdges != null && edge.source() != edge.dest())
        inEdges.put(edge.dest(), edge);
}
```

Correctness Highlights: By INEDGES, the instance variable *inEdges* remains *null* if the graph is directed, or if inedges are not to be supported. When the inedges are kept, inserting *edge* into the *inEdges* for *edge.dest* preserves INEDGES. ■

The public *addEdge* method takes *edge*, the edge to add to this graph.

```
public void addEdge(E edge) {
    addEdge(edge.source(), edge);
    if (!isDirected)
        addEdge(edge.dest(), edge);
}
```

Correctness Highlights: Suppose the graph is directed, and let $edge = (u, v)$. By the correctness of the internal *addEdge* method, *edge* is added to the outedges of *u* to preserve DIRECTED-GRAPH. Also, the internal *addEdge* method preserves INEDGES.

Next, we consider when the graph is undirected. Let $edge = \{u, v\}$. By the correctness of the internal *addEdge* method, *edge* is added to the outedges of *u*, and *edge* is added to the outedges of *v*. Thus, UNDIRECTEDGRAPH is preserved. ■

The *removeVertex* method that takes *vertex*, the vertex to remove. It returns *true* if the vertex was in the graph, and *false* otherwise.

```
public boolean removeVertex(V vertex) {
    if (vertices.contains(vertex)) {
        vertices.remove(vertex);
        if (outEdges.contains(vertex))
            outEdges.removeBucket(vertex);
        if (inEdges != null && inEdges.contains(vertex))
            inEdges.removeBucket(vertex);
        return true;
    }
    return false;
}
```

Correctness Highlights: By the correctness of *contains*, the correct value is returned. We now argue that when *vertex* is in the graph that it is removed. *VERTICES* is preserved by removing *vertex* from *vertices*. As discussed, in the optimizations, our implementation uses a lazy delete of the edges. The bucket in *outEdges*, and *inEdges* when maintained, can be removed. However, edges in other lists that reference an edge with that vertex, are removed during iteration. ■

The internal *removeEdge* method takes *vertex*, the vertex that is to be treated as the source vertex of *edge*, and *edge*, the edge to remove. It returns *true* when the edge is successfully removed, and *false*, if the edge did not exist.

```
boolean removeEdge(V vertex, E edge) {
    if (outEdges.remove(vertex, edge)) {
        if (inEdges != null)
            inEdges.remove(edge.dest(), edge);
        return true;
    }
    return false;
}
```

Correctness Highlights: By the correctness of the bucket mapping *remove* method, the edge is removed from *outEdges*. By *INEDGES*, the instance variable *inEdges* is only *null* when the incoming edges are maintained. By removing the edge from *inEdges*, *INEDGES* is preserved. Finally, observe that *true* is returned exactly when the edge was in the graph. ■

The public *removeEdge* method takes *edge*, the edge to be removed from this graph. It returns *true* when the edge is successfully removed, and *false*, if the edge did not exist.

```
public boolean removeEdge(E edge) {
    if (removeEdge(edge.source(), edge)) {
        if (!isDirected && edge.source() != edge.dest())
            return removeEdge(edge.dest(), edge);
        return containsVertex(edge.dest());
    }
    return false;
}
```

//returns true if edge was in list
//check that dest is in the graph

Correctness Highlights: By the correctness of the internal *removeEdge* method, the edge is removed if it exists in the graph. Furthermore, when the edge is in the graph, removing the edge from the destination vertex to the source vertex in an undirected graph when *edge.source* $\neq$ *edge.destination* preserves *UNDIRECTEDGRAPH*. ■

55.4 Edge Iterators

The *edgesFrom* method that takes *source*, the source vertex, returns an iterator over the outgoing edges from the given vertex. Observe that for an undirected graph, the iterator is over all edges incident to the given vertex.

```
public Iterator<E> edgesFrom(V source) {
    return new EdgeIterator(outEdges.getElements(source));
}
```

The *edgesTo* method that takes *dest*, the destination vertex, returns an iterator over the incoming edges from the given vertex. Observe that there is no need to maintain the *inEdges* in an undirected graph, since this method can be supported by the *edgesFrom* method. This method is optional. It throws an *UnsupportedOperationException* when the graph is directed and the constructor parameter specified not to maintain the *inedges*.

```
public Iterator<E> edgesTo(V dest) {
    if (!isDirected)
        return edgesFrom(dest);
    if (inEdges == null)
        throw new UnsupportedOperationException("incoming edges are not stored");
    return new EdgeIterator(inEdges.getElements(dest));
}
```

55.5 Edge Iterator Inner Class

EdgeIterator implements *Iterator<E>*

We wrap the edge iterator to implicitly remove edges incident to vertices that have been deleted. To save the overhead of wrapping the iterators, one could instead store all outedges and inedges in a single cross linked data structure (see optimizations on [page 904](#)), and remove all of the incident edges of a vertex immediately when that vertex is removed.

```
Locator<E> wrapped;           //the wrapped locator
E lastEdgeReturned;          //the edge most recently returned by this iterator
```

The constructor takes *edges*, a locator over a set of edges that is used to initialize the value for the wrapped locator.

```
EdgeIterator(Locator<E> edges) {
    this.wrapped = edges;
}
```

The *hasNext* method moves to the next edge, and deletes any edges that were implicitly removed when the destination vertex (or the source vertex in an undirected graph) was removed. While this method can take time proportional to the number of edges in the particular list being considered, the amortized cost is constant, since each lazily deleted edge is considered only one time, so the cost of the remove can be “charged” to the corresponding call of the public method *addEdge*.

```
public boolean hasNext() {
    while (wrapped.hasNext()) { //more edges in the list
        wrapped.advance(); //move to the next edge
        E edge = wrapped.get();
        if (!vertices.contains(edge.dest()) //if destination vertex no longer exists
            || !vertices.contains(edge.source())) //or source vertex no longer exists
            wrapped.remove(); //remove the edge
        else {
            wrapped.retreat(); //move back so next advance behaves properly
            return true;
        }
    }
    return false;
}
```

Correctness Highlights: The *hasNext* method is where we resolve the lazy delete performed when a vertex is deleted. By *UNDIRECTED*, it is possible that the edge with source vertex *u* and destination vertex *v* is in the outgoing list of edges for *v*. If *u* is deleted, then the edge in the outgoing list for *v* will still be in the list, but the iterator should skip it because it is not really in the graph. As an optimization, whenever such a removed edge is encountered it is removed from the list of edges. Observe that a similar behavior can occur when an edge with source vertex *u* and destination vertex *v* is in the list of incoming edges for *u*, and then *v* is deleted. Finally, observe, that *hasNext* should not change the position of the iterator. Thus, if an edge is removed, the wrapped locator must be moved back to previous edge. Furthermore, the process must be repeated from the locator’s new position, since it is not yet known if there is another edge in the list that remains in the graph. ■

The *next* method moves the locator to the next edge in the list, which is in the graph. It throws a *NoSuchElementException* when there is no next edge in the iteration.

```
public E next() {
    if (!hasNext())
        throw new NoSuchElementException();
    return lastEdgeReturned = wrapped.next();
}
```

The *remove* method removes the edge most recently returned by this iterator. The method must preserve *DIRECTED* and *UNDIRECTED*, which is accomplished by calling the *removeEdge* method on the last edge returned. This method throws a *NoSuchElementException* when *next* has not been called since the most recent call to *remove*.

```

    public void remove() {
        if (lastEdgeReturned == null)
            throw new NoSuchElementException();
        removeEdge(lastEdgeReturned);
        lastEdgeReturned = null;
    }
}

```

55.6 Adjacency List Class

package graph

AbstractGraph<V, E extends Edge<V>> implements *Graph*<V,E>

└─ **AdjacencyList**<V,E extends Edge<V>> implements *Graph*<V,E>

The AdjacencyList class just contains two constructors, that create a new adjacency list representation. The constructor can specify whether or not the graph is directed, whether it is a multigraph, and whether the adjacency list should store the incoming edges for each vertex.

```

    public AdjacencyList(boolean directed, boolean containsMultiEdges,
                        boolean storeIncomingEdges) {
        super(new AdjacencyListRepresentation<V, E>(directed, storeIncomingEdges,
                                                    containsMultiEdges);
    }

```

The constructor with no arguments uses the default values of having a graph representation for a directed graph that is a directed multigraph that maintains a list of incoming edges for each vertex.

```

    public AdjacencyList() {
        this(true, true, true);
    }

```

55.7 Performance Analysis

The asymptotic time complexities of the graph representation methods for an adjacency list, augmented adjacency list, adjacency set, and augmented adjacency set are given in [Tables 55.4](#) and [55.5](#), and the asymptotic time complexities for the edge iterator methods are given in [Table 55.6](#). The tables are based on the assumption that multi-edges are permitted. When multi-edges are not allowed, the time complexity for *addEdge* is the same as that of *containsEdge*.

The *isDirected*, *allowsMultiEdges*, and *numVertices* methods just return the value of an instance variable, so they take constant time. Likewise, *iterator* takes constant time since it just initializes an iterator over the set *vertices*. The *edgesFrom* and *edgesTo* methods take constant time to initialize the incident edge iterator.

method	Time Complexity		
	AdjacencyList (Directed)	AdjacencyList (Undirected)	AdjacencyList (Augmented)
<i>addEdge(e)</i>	$O(1)$	$O(1)$	$O(1)$
<i>addVertex(v)</i>	$O(1)$	$O(1)$	$O(1)$
<i>containsVertex(v)</i>	$O(1)$	$O(1)$	$O(1)$
<i>edgesFrom(v)</i>	—	—	$O(1)$
<i>edgesTo(v)</i>	$O(1)$	$O(1)$	$O(1)$
<i>isDirected()</i>	$O(1)$	$O(1)$	$O(1)$
<i>isMultiGraph()</i>	$O(1)$	$O(1)$	$O(1)$
<i>iterator()</i>	$O(1)$	$O(1)$	$O(1)$
<i>numVertices()</i>	$O(1)$	$O(1)$	$O(1)$
<i>removeVertex(v)</i>	$O(1)$	$O(1)$	$O(1)$
<i>containsEdge(u, v)</i>	$O(d_{\text{out}}(u))$	$O(\min(d_{\text{out}}(u), d_{\text{in}}(v)))$	$O(\min(d_{\text{out}}(u), d_{\text{in}}(v)))$
<i>getEdge(u, v)</i>	$O(d_{\text{out}}(u))$	$O(\min(d_{\text{out}}(u), d_{\text{in}}(v)))$	$O(\min(d_{\text{out}}(u), d_{\text{in}}(v)))$
<i>removeEdge(u, v)</i>	$O(d_{\text{out}}(u))$	$O(d_{\text{out}}(u) + d_{\text{out}}(v))$	$O(\min(d_{\text{out}}(u) + d_{\text{in}}(v)))$
<i>edgeSet()</i>	$O(n + m)$	$O(n + m)$	$O(n + m)$

Table 55.4 Summary of the asymptotic time complexities for maintaining a graph using an adjacency list and augmented adjacency list. For vertex v , $d_{\text{out}}(v)$ is the outdegree of v and $d_{\text{in}}(v)$ is the indegree of v . A dash indicates that the method is unsupported.

method	Time Complexity		
	AdjacencySet (Directed)	AdjacencySet (Undirected)	AdjacencySet (Augmented)
<i>addEdge(e)</i>	$O(1)$	$O(1)$	$O(1)$
<i>addVertex(v)</i>	$O(1)$	$O(1)$	$O(1)$
<i>containsEdge(u, v)</i>	$O(1)$	$O(1)$	$O(1)$
<i>edgesFrom(v)</i>	—	—	$O(1)$
<i>edgesTo(v)</i>	$O(1)$	$O(1)$	$O(1)$
<i>getEdge(u, v)</i>	$O(1)$	$O(1)$	$O(1)$
<i>isDirected()</i>	$O(1)$	$O(1)$	$O(1)$
<i>isMultiGraph()</i>	$O(1)$	$O(1)$	$O(1)$
<i>iterator()</i>	$O(1)$	$O(1)$	$O(1)$
<i>numVertices()</i>	$O(1)$	$O(1)$	$O(1)$
<i>removeEdge(u, v)</i>	$O(1)$	$O(1)$	$O(1)$
<i>removeVertex(v)</i>	$O(1)$	$O(1)$	$O(1)$
<i>edgeSet()</i>	$O(n + m)$	$O(n + m)$	$O(n + m)$

Table 55.5 Summary of the asymptotic time complexities for maintaining a graph using an adjacency set and augmented adjacency set. For vertex v , $d_{\text{out}}(v)$ is the outdegree of v and $d_{\text{in}}(v)$ is the indegree of v . A dash indicates that the method is unsupported.

edge iterator method	average time complexity
constructor	$O(1)$
<i>advance()</i> when preceded by <i>hasNext</i>	$O(1)$
<i>next()</i> when preceded by <i>hasNext</i>	$O(1)$
<i>advance()</i>	$O(1)$, amortized
<i>hasNext()</i>	$O(1)$, amortized
<i>next()</i>	$O(1)$, amortized
<i>remove()</i>	$O(T_R(G))$

Table 55.6 Summary of the time complexities for the incident edge iterator methods for an adjacency list. We use d to denote the number of edges in the iteration sequence for this iterator. For example, if the iterator is created by *edgesFrom*(v), then d is the outdegree of v . Likewise, if the iterator is created by *edgesTo*(v), then d is the indegree of v . We use $T_R(G)$ for the time complexity to remove an edge from a graph G using the graph representation *removeEdge* method.

It takes expected constant time to add a vertex to the graph, the time to insert it into *vertices*. Since it takes constant expected cost to determine if a vertex is in *vertices*, the expected time complexity for *containsVertex* is also constant. To remove a vertex from the graph it takes expected constant time to remove the vertex from *vertices*, and constant time to remove it from *outEdges* and *inEdges* (when maintained).

We now consider the cost of adding an edge $e = (u, v)$ to a directed graph. It takes worst-case constant time to insert e into $u.outEdges$ when it is an adjacency list. For an augmented adjacency list it also takes constant time to insert e into $v.inEdges$. For an undirected graph, this cost is doubled since each edge can be viewed being inserted as two directed edges. So for both an adjacency list and augmented adjacency list, adding an edge takes worst-case constant time. For an adjacency set and an augmented adjacency set, the only difference is that it takes constant expected time to insert an edge into *outEdges* and *inEdges*. So for both an adjacency set and augmented adjacency set, adding an edge takes expected constant time.

We now analyze the cost of finding the edge $e = (u, v)$ for a directed graph. For an adjacency list the cost is $O(d_{out}(u))$ to traverse $u.outEdges$. For an augmented adjacency list, the smaller of $u.outEdges$ and $v.inEdges$ can be traversed, leading to a cost of $O(\min(d_{out}(u), d_{in}(v)))$. For an adjacency set and augmented adjacency set, the edge can be located in expected constant time. The costs of the *containsEdge* and *getEdge* methods are dominated by the cost of locating the edge.

Next, we consider *removeEdge*. For an undirected graph in which the incoming edges are kept, to remove edge $e = (u, v)$ it must be located in both $u.outEdges$ and $v.inEdges$ which has time complexity $O(d_{out}(u) + d_{in}(v))$. For a directed graph, it takes constant time to remove the edge once located. However, for an undirected graph, since $e = \{u, v\}$ is in the outedges for both u and v , it must be located in both lists, leading to a time complexity $O(d_{out}(u) + d_{out}(v))$.

The *edgeSet* method iterates through the outgoing edges from all vertices, which has worst-case cost $O(n + m)$ for both an adjacency list and augmented adjacency list and expected cost $O(n + m)$ for both an adjacency set and augmented adjacency set.

We now analyze the time complexity of the incident edge iterator methods. The constructor clearly takes constant time. Observe that for all other methods, except for edges lazily deleted, the next list item is next in the iteration order. Thus *advance*, *hasNext*, and *next* have amortized constant cost. Since *hasNext* advances the iterator to just before the next edge, if *advance* or *next* is called after *hasNext*, it has worst-case constant cost.

55.8 Quick Method Reference

AdjacencyListRepresentation Public Methods

- p. 906** *AdjacencyListRepresentation*(boolean isDirected, boolean storeIncomingEdges)
- p. 908** void **addEdge**(E edge)
- p. 908** boolean **addVertex**(V vertex)
- p. 907** boolean **containsEdge**(V source, V dest)
- p. 906** boolean **containsVertex**(V vertex)
- p. 910** Iterator<E> **edgesFrom**(V source)
- p. 910** Iterator<E> **edgesTo**(V dest)
- p. 907** E **getEdge**(V source, V dest)
- p. 906** boolean **isDirected**()
- p. 908** NonMutatingIterator<V> **iterator**()
- p. 906** int **numVertices**()
- p. 909** boolean **removeEdge**(E edge)
- p. 909** boolean **removeVertex**(V vertex)

AdjacencyListRepresentation Internal Methods

- p. 908** void **addEdge**(V vertex, E edge)
- p. 909** boolean **removeEdge**(V vertex, E edge)

Chapter 56

Weighted Graph ADT

A weighted graph represents general relationships between pairs of elements from among a set of elements, where each relationship has an associated real-valued “weight.” Many practical algorithmic problems can be formulated most naturally as weighted graph problems. Examples include the arbitrage problem discussed in Section 2.1, the airline travel agent application described in Section 56.1, and the image segmentation problem discussed in Section 56.2.

The algorithms that perform computations on weighted graphs are presented and implemented in terms of the *WeightedGraph* interface, independent of the graph representation. So a graph algorithm need not be reimplemented for each representation. The *WeightedGraph* interface extends the *Graph* interface, so algorithms defined for unweighted graphs are also available to applications using a weighted graph.

56.1 Case Study: Airline Travel Agent (Revisited)

Consider a simple automated travel agent that has access to information about airports and flights. For the set of airports A , information about each airport $a \in A$ includes a name, a city, a three letter acronym (e.g., STL for St. Louis, LAX for Los Angeles), the time zone in which the airport is located, a longitude, and a latitude. The set of commercial flights F includes flights that depart from some airport in A and arrives at some airport in A . Information about each flight $f \in F$ includes an airline, a flight number, a departure airport (specified using the three letter acronym), a departure time (in the local time zone of the departure airport), an arrival airport (specified using the three letter acronym), an arrival time (in the local time zone of the arrival airport), and other information about the fare, availability, etc. We will assume that this information is in a publicly accessible database, and its form cannot be changed by this application.

We consider the following problem. A traveler provides the acronym of a departure airport, a time (in the time zone of the departure airport) when the traveler plans to be at the departure airport, and the acronym for a destination airport. The problem is to find a sequence of flights that will get the traveler to the destination airport at the earliest time possible.

First, observe that we can view a flight as a relationship between airports. Thus, we can model this problem as a graph problem in which there is a vertex corresponding to each airport, and an edge corresponding to each flight. The problem reduces to finding a path in the graph from the departure vertex to the arrival vertex that minimizes the elapsed time.

We create an *Airport* class that has an instance variable for each relevant airport attribute. One instance variable will be the three letter acronym that is used as a key to identify the airport. Since the user will specify the acronym to identify an airport, a mechanism is required to retrieve the airport instance associated with the provided acronym. As discussed in Section 20.1, when this case study was first introduced, one option is to store the airport instances in a set, where an equivalence tester is defined that deems two airports equivalent if they have the same three letter acronym.

The *getEquivalentElement* method could be used to retrieve the airport by using a “target airport” instance in which the acronym is set to the desired value, and all other attributes are *null*. While this approach is the most space efficient, a more natural approach for this application (at the cost of some additional space) would be to store the airports using a mapping where the key is the three letter acronym, and the airport instance is the associated data. Using this approach, we would incur the space overhead of creating a tagged element for each airport.

We also introduce a *Flight* class that implements the *WeightedEdge* interface. For each flight, the departure city is the source vertex, and the arrival city is the destination vertex. Since the goal is to minimize travel time, the natural value for the edge weight of a flight is the scheduled duration of the flight. (By using the mapping from acronyms to airports, the time zone for both the departure and arrival cities can be obtained and used to compute the duration of the flight once, as each edge is added to the graph.)

Observe that the resulting graph is a directed weighted multigraph. The edges are directed since each flight goes from the departure airport to the arrival airport, and the edges are weighted by the flight duration. We use a multigraph since typically there are many flights between a pair of airports. Furthermore, unlike more typical shortest path problems, it is important to maintain retain all edges between two vertices. For example there might be a 2 hour flight from airport A to airport B, and a 2.5 hour flight from airport A to airport B. If these flights leave at the same time then an optimal solution would never use the 2.5 hour flight. However, suppose the traveler arrives at airport A at 6:00pm, the 2 hour flight leaves at 9:00pm, and the 2.5 hour flight leaves at 7:30pm. The 2.5 hour flight is scheduled to get the traveler to city B an hour sooner, so it is the better choice.

Having defined the graph, we consider algorithms for solving the problem. There is a starting airport and a starting time, and a specified destination that the traveler wants to reach in the shortest time possible. At first glance, this seems very close to the single-source shortest path problem where the starting airport is the source. Once the shortest path tree is computed, a shortest path (i.e., a sequence of flights to take) to all other airports, including the specified destination, can be obtained. However, there is one significant difference. This is not the classic single-source shortest path problem, where the weight of the path is the sum of the edge weights along the path. For this application, that solution would correspond to the time spent on all of the flights, and completely neglect the time spent in airports during each layover. As illustrated by the above 2.0 and 2.5 hour flights example, if the time spent at airports is ignored, then the solution computed would be different.

Since there are no negative weight edges (and the layover time is also non-negative), we can use Dijkstra’s shortest path algorithm (Section 57.2) as a starting point. However, we must modify it to incorporate a cost at each airport that corresponds to the layover time. Let v be an intermediate airport. In Dijkstra’s shortest path algorithm the cost associated with v is the sum of the edge weights in a shortest path from the source to v . In our application, that would correspond to the sum of the flight lengths used to get from the originating airport to v . For this application, however, the cost associated with v should be the elapsed time between arrival at the originating airport, and arrival at v . Observe that if v is reached using flight f from airport u , then the cost associated with v is the cost associated with u , plus the layover time at airport u , plus the length of flight f . Observe that this cost can be computed using knowledge about airport u , flight f , and airport v .

So, we can obtain a solution to our problem by defining an application-specific *getCost* function for the greedy tree builder algorithm (Section 57.1) that incorporates the layover cost as described above. To design this solution, it is important to understand the greedy tree builder and how the cost function, which determines the priority of a vertex, can be modified to solve different problems. However, to use a “black box” implementation of Dijkstra’s algorithm (or any standard single-source shortest path algorithm without modification), the solution would be to construct a second graph with $d_{\text{in}}(v) \cdot d_{\text{out}}(v)$ vertices for each original vertex v , where each of these vertices corresponds to a different combination of inbound and outbound flights at v . Since each of these new vertices would have only one incoming and one outgoing flight, the layover time for that combina-

tion of flights could be added to the weight of the outgoing edge. Finally, one would need a way to treat all of these vertices as the same if they were the originating or destination airport. Clearly, the flexibility of the greedy tree builder provides a much more satisfactory solution, in the space usage, time complexity, and simplicity of the code.

56.2 Case Study: Image Segmentation

An important problem in the area of computer vision is **image segmentation**, the task of partitioning an image into its components that ideally correspond to the objects in the image. In this case study, we focus on the simpler task of **object segmentation**, in which the image contains a picture of an object, and the goal is to partition the image into two portions: the **foreground** (pixels that are part of the object) and the **background** (pixels that are not part of the object). Object segmentation is also known as object/background segmentation, figure/background segmentation, and foreground/background segmentation.

A digital image is a two-dimensional grid of pixels, called a **raster**, in which each pixel has three color values (e.g., red, green, blue) typically in the range 0 to 255*. We assume that a **similarity measure** between each pair of pixels has been defined. For example, one could view each pixel as a point in a three-dimensional space (as defined by its three color values) and measure the similarity of two pixels as the Euclidean distance between them in this three-dimensional space. Note that this is different from the distance between the points in the raster.

The goal of object segmentation is to partition the pixels of the image into two sets: F , the set of pixels in the foreground, and B the set of pixels in the background. Intuitively, the property desired of this partition is that the pixels in F are more similar to each other than to any pixels in B . More specifically, it is desirable for neighboring and visually similar pixels to have the same label. Therefore, a penalty is also assessed when two adjacent pixels that are visually similar have different labels. We now provide a mechanism to formalize this intuition. We define a **discontinuity penalty** c_d for assigning visually similar pixels p_i and p_j to *different* components. Any function can be used, and in general it need not be symmetric. As a simple example, one possibility is to let $c_d = e^{-dist(p_i, p_j)}$ where e is Euler's constant, and $dist$ is the similarity measure between p_i and p_j . Observe that $0 \leq c_d(p_i, p_j) \leq 1$, for each pair of pixels p_i, p_j . When p_i and p_j are very similar, $c_d(p_i, p_j)$ is near 1 (high penalty), and when p_i and p_j are very different, $c_d(p_i, p_j)$ is near 0 (low penalty).

The **label penalty** c_ℓ corresponds to the penalty associated with assigning a pixel to the wrong partition (i.e., to assign a pixel in the foreground to the background, or to assign a pixel in the background to the foreground). Often the label penalty is based on a combination of an application specific likelihood function that gives the probability that a pixel is in the foreground (e.g., based on its location in the image). Often, features of the pixel (e.g., color) are combined with the likelihood function to determine the label penalty. We use $c_\ell^{(B)}(p_i)$ to denote the label penalty for incorrectly assigning p_i to the background, and $c_\ell^{(F)}(p_i)$ to denote the label penalty for incorrectly assigning p_i to the foreground. Let $L = \{F, B\}$ be the set of labels, let P be the set of pixels, and let N be the set of pairs of adjacent pixels (neighbors). Let $L(p_i, p_j) = 1$ when p_i and p_j are in different components, and let $L(p_i, p_j) = 0$ when p_i and p_j are in the same component. The exact definition of when two pixels are adjacent can be defined for the specific application. The goal is to select the partition L that minimizes

*Alternatively the grid could be defined over groups of pixels that have both color and texture features. For simplicity, we use a grid of pixels in our discussion here.

$$E(L) = \sum_{p_i, p_j \in N} c_d(p_i, p_j) L(p_i, p_j) + \sum_{p \in B} c_\ell^{(B)}(p) + \sum_{p \in F} c_\ell^{(F)}(p).$$

The first term captures the discontinuity penalty, and the last two terms capture the penalty for any incorrectly placed pixels.

We now formulate, as a graph problem, the task of finding a partition L that minimizes $E(L)$. Let $G = (V, E)$ be a directed weighted graph. We introduce a vertex v_p for each $p \in P$, and a vertex f to represent the foreground, and a vertex b to represent the background. So $|V| = |P| + 2$. We assume some pixel $p_f \in P$ is specified to be in the foreground. This pixel could be specified by a user, or perhaps in a specified location such as the center of the image. For example, if the goal is to segment a tumor from the rest of a medical image, a physician might select some point on the tumor to identify p_f . Similarly, some pixel $p_b \in P$ is specified to be in the background.

We introduce the following edges E into the graph, where the weight of each edge (v_i, v_j) is assigned to capture the “value” of placing p_i and p_j together in the same component (F or B). For each pair of pixels (p_i, p_j) in N , we place an edge from v_i to v_j with weight $c_d(p_i, p_j)$, and an edge from v_j to v_i with weight $c_d(p_j, p_i)$. Also, we want to capture that the “value” of putting a pixel in the foreground saves us from the penalty that would otherwise be incurred if that pixel were placed in the background. Therefore, for each vertex $v_i \in V - \{v_f, v_b\}$ we place an edge from f to v_i with weight $c_\ell^{(B)}(p)$, and similarly from v_i to b with weight $c_\ell^{(F)}(p)$. Finally, we place an edge from f to v_f with infinite weight, and an edge from v_b to b with infinite weight. This captures a strong “bond” between v_f and the special foreground vertex, and a strong bond between v_b and the special background vertex.

The problem that remains is to compute $L = \{F, B\}$ that minimizes $E(L)$. We achieve this goal by partitioning all vertices in the graph into two sets V_F, V_B where $f \in V_F$ and $b \in V_B$. The pixels corresponding to vertices in $V_F - \{f\}$ will define F , and the pixels in $V_B - \{b\}$ will define B . Let

$$E_{(F,B)} \subset E = \{e = (u, v) \in E \mid u \in V_F \wedge v \in V_B\}.$$

Let $w(E_{F,B}) = \sum_{e \in E_{(F,B)}} e.\text{weight}$. So $w(E_{F,B})$ is the sum of the edge weights over all edges in which the source is in V_F and the destination is in V_B . Observe that if one were to separate (or cut) the graph into V_F and V_B , the edges in $E_{(F,B)}$ would be cut. So $E_{(F,B)}$ is often called a **cut**. The choice of V_F, V_B that will minimize $E(L)$ is exactly the cut with minimum weight. The first term in $E(L)$ corresponds to the edges cut where the source is a pixel in V_F and the destination is a neighboring pixel in V_B , the second term in $E(L)$ corresponds to the edges cut where the source is f and the destination is a pixel in V_B , and finally the third term in $E(L)$ corresponds to the edges cut where the source is in V_F and the destination is b . By construction f is placed in V_F and b is placed in V_B (since there would be an infinite weight edge cut otherwise).

The algorithmic question is how to partition the vertices of a directed weighted graph, so the sum of the edge weights in the cut is minimized. This **minimum cut problem** is equivalent to the classic **maximum flow problem** in which f is called the **source**, and b is called the **sink**. Section 57.8 discusses the maximum flow problem and presents the Edmonds-Karp algorithm to efficiently find an optimal solution. The Edmonds-Karp maximum flow algorithm (or any desired maximum flow algorithm) can be used to find a minimum cut, which in turn defines the desired segmentation. Greig, Porteous, and Seheult [78] first used this graph-based approach in 1989 to perform object segmentation. Recently there has been a variety of image segmentation algorithms, and algorithms for related problems that formulate the segmentation problem as the problem of finding a minimum cut in a graph [128, 30, 29]. The maximum flow problem can be generalized to partition the graph into k regions by introducing k **terminal vertices** (versus just f and b). For general image segmentation, there would be one terminal vertex per segment. The terminal vertices can also be associated with other features. For example, for the stereo correspondence problem, the depth is used [128].

56.3 Terminology

Throughout this chapter, we use the terminology defined for an unweighted graph (Chapter 52), and the following additional terminology.

- We use **unweighted shortest path** to refer to the shortest path when weights are ignored, and measured only by the number of edges on the path.
- For edge $e \in E$, we use $e.weight$ to denote the weight of edge e .
- The **weight** of a collection of edges $E' \subset E$ is $w(E') = \sum_{e \in E'} e.weight$. For example, for path $P = \langle e_1, e_2, \dots, e_k \rangle$ has weight $w(P) = \sum_{i=1}^k e_i.weight$.
- A **negative weight cycle** is defined as a cycle C in a weighted graph for which $w(C) < 0$.
- A **shortest path** in a weighted graph from vertex u to v is defined as a path P from u to v such that for any other path P' from u to v , $w(P) \leq w(P')$.
- If there is a negative weight cycle C reachable from s , then we say that the shortest path from s to any other vertex is **undefined** since the weight of the path can go to $-\infty$ by repeatedly traversing cycle C .
- An algorithm solving the **single-source shortest path problem** for a weighted graph takes as input a specified source vertex s , and computes a shortest path tree for s (unless there is a negative weight cycle reachable from s).
- A **shortest path matrix** maintains within it a shortest path tree for each vertex of the graph. Thus, a shortest path matrix stores a shortest path between all pairs of vertices in the graph.
- An algorithm solving the **all-pairs shortest path problem** computes a shortest path matrix (unless there is a negative cycle in the graph).
- A **spanning tree** of an undirected graph $G = (V, E)$ is a set of $|V| - 1$ edges $T \subseteq E$ that connect all vertices. That is, $G_T = (V, T)$ is acyclic (i.e., a tree) and has a single connected component (so it spans all vertices).
- A **minimum spanning tree** of an undirected weighted graph $G = (V, E)$ is a spanning tree T of G such that for any other spanning tree T' of G , $w(T) \leq w(T')$.
- A **flow network** is a weighted directed graph without any self loops in which all edge weights are non-negative. The weight on edge e corresponds to a **capacity** of e , the amount of flow that can be routed along e . A flow network specifies vertex $s \in V$ as the **source** and vertex $t \in V$ as the **sink**. A **flow** in the graph is a function $f : E \rightarrow \mathbb{R}$.
- A **legal flow** f is a flow in which for all $e \in E$, $f(e) \leq e.weight$, and for all $v \in V - \{s, t\}$,

$$\left(\sum_{e \in v.inedges} f(e) \right) = \left(\sum_{e \in v.outedges} f(e) \right).$$

- We define the value for flow f as $|f| = \sum_{e \in s.outedges} f(e)$.
- A legal flow f is a **maximum flow** if for any other legal flow f' , $|f| \geq |f'|$.

- For edge e in a flow network, we say that e is **saturated** when $f(e) = e.weight$.
- A **cut** of a graph $G = (V, E)$ is a partition of the vertices into (V_1, V_2) . That is $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \emptyset$.
- Let $E_{(V_1, V_2)} = \{e = (u, v) \in E \mid u \in V_1 \wedge v \in V_2\}$.
- The **capacity** $w(C)$ of cut $C = (V_1, V_2)$ in a directed graph $G = (V, E)$ is $w(C) = \sum_{e \in E_{(V_1, V_2)}} e.weight$.
- An (S, T) -**cut** is a cut for which $s \in S, t \in T$. A **minimum cut** of a flow network is an (S, T) cut C such that for any other (S, T) cut C' , $w(C) \leq w(C')$.

Section 52.6 discusses the trade-offs between the adjacency matrix and adjacency list representation. The only required change in the representation of a weighted graph is to extend the *Edge* interface to define a *WeightedEdge* (Section 56.4). The weighted adjacency matrix (Section 56.8) and the weighted adjacency list (Section 56.9), need only define the constructors that call the corresponding constructors in the unweighted representations they extend.

56.4 Weighted Edge Interface

For both the adjacency matrix and adjacency list graph representations, the only change needed for representing a weight graph is to extend the *Edge* interface to a *WeightedEdge*.

```
public interface WeightedEdge<V> extends Edge<V> {
    double weight();                                     //returns the weight of this edge
}
```

56.5 Simple Weighted Edge Class

The simple weighted edge class provides a sample implementation for a weighted edge. The flow graph returned by the maximum flow algorithm uses simple weighted edges.

```
public class SimpleWeightedEdge<V> extends SimpleEdge<V>
    implements WeightedEdge<V> {
    double weight;

    public SimpleWeightedEdge(V head, V tail, double weight){
        super(head, tail);
        this.weight = weight;
    }
    public double weight() {return weight;}
}
```

56.6 Weighted Graph Interface

The weighted graph algorithms we present do not depend on a particular graph implementation. Instead, they are implemented entirely in terms of the WeightedGraph ADT. This allows the presentation and implementation of the graph algorithms to be independent of the underlying graph representation. Typically an ADT does not specify the algorithms that an implementation should use to support its methods. However, for the WeightedGraph ADT there is a tight coupling between the interface and the underlying algorithms, so for clarity we specify these algorithms here.

WeightedGraph(GraphRepresentation<V,E> rep, boolean directed, boolean multigraph):

Creates a new empty weighted graph that uses the given graph representation. The second and third parameters specify whether or not the graph is directed, and whether or not it is a multigraph.

ShortestPathMatrix<V,E> allPairsShortestPaths(): Uses the Floyd-Warshall all-pairs shortest path algorithm to compute and return a shortest path matrix.

InTree<V,E> generalShortestPathFromSource(V source): Uses the Bellman-Ford shortest path algorithm to compute and return a shortest path tree for the given source vertex. This algorithm will yield the correct answer if there are negative weight edges, provided that there is no negative weight cycle.

Set<E> kruskalMST(): Uses Kruskal's minimum spanning tree to return a set of edges that form a minimum spanning tree for this graph. It throws a *DisconnectedGraphException* when the graph is not connected, a *NegativeWeightEdgeException* when a negative weight edge is encountered, and a *GraphException* when the graph is directed.

FlowGraph maximumFlow(V source, V sink): Uses the Edmonds-Karp implementation of the basic Ford-Fulkerson augmenting path method to find a maximum flow (and a minimum cut) for the flow network defined by the provided source and sink.

Set<E> primMST(): Uses Prim's minimum spanning tree to return a set of edges that forms a minimum spanning tree for this graph. It throws a *DisconnectedGraphException* when the graph is not connected, a *NegativeWeightEdgeException* when a negative weight edge is encountered, and an *IllegalArgumentException* when the graph is directed.

InTree<V,E> weightedShortestPaths(V source): Uses Dijkstra's shortest path algorithm to compute and return a shortest path tree for the given source vertex. It throws a *NegativeWeightEdgeException* when the graph has a negative weight edge reachable from the source.

56.7 Selecting a Data Structure

The process of selecting an adjacency matrix versus an adjacency list representation for a weighted graph is the same as the selection process described in Section 52.6 for an unweighted graph.

56.8 Weighted Adjacency Matrix Data Structure

package graph

AbstractGraph<V, E extends Edge<V>> implements *Graph*<V,E>

└ AbstractWeightedGraph<V, E extends WeightedEdge<V>> implements *WeightedGraph*<V,E>

└ **WeightedAdjacencyMatrix**<V,E extends **WeightedEdge**<V>>
implements *WeightedGraph*<V,E>

```
public WeightedAdjacencyMatrix(boolean directed, boolean multigraph) {  
    super(new AdjacencyMatrixRepresentation<V, E>(directed), multigraph);  
}
```

```
public WeightedAdjacencyMatrix() {  
    this(true, false);  
}
```

56.9 Weighted Adjacency List Data Structure

package graph

AbstractGraph<V, E extends Edge<V>> implements *Graph*<V,E>

└ AbstractWeightedGraph<V, E extends WeightedEdge<V>> implements *WeightedGraph*<V,E>

└ **WeightedAdjacencyList**<V,E extends **WeightedEdge**<V>>
implements *WeightedGraph*<V,E>

```
public WeightedAdjacencyList(boolean directed, boolean multigraph,  
                             boolean storeIncomingEdges) {  
    super(new AdjacencyListRepresentation<V, E>(directed, storeIncomingEdges),  
        multigraph);  
}
```

```
public WeightedAdjacencyList() {  
    this(true, false, true);  
}
```

Chapter 57

Abstract Weighted Graph and Weighted Graph Algorithms

package graph

AbstractGraph<V, E extends Edge<V>> implements Graph<V,E>

↑ **WeightedGraph**<V, E extends WeightedEdge<V>> implements **Graph**<V,E>

In this chapter, we present algorithms to perform computations on a weighted graph in terms of the *WeightedGraph* interface. This allows any graph representation to be used. We include the following algorithms:

Greedy Tree Builder: We present a general greedy algorithm for constructing a tree. This algorithm can be instantiated to solve a variety of problems including the single-source shortest path problem, the minimum spanning tree problem, and the maximum bottleneck problem (page 927). When using an adjacency list for the graph representation and using a Fibonacci heap (Chapter 28) for the tagged priority queue, the tree builder method has worst-case asymptotic time complexity $O(m + n \log n)$.

Dijkstra's Single-Source Shortest Path Algorithm: This algorithm computes a shortest path tree for a weighted graph in which there are no negative weight edges. Dijkstra's algorithm uses the greedy tree builder and has worst-case asymptotic time complexity $O(m + n \log n)$.

Prim's Minimum Spanning Tree Algorithm: This algorithm computes a minimum spanning tree for a weighted graph that has no negative weight edges. Prim's algorithm uses the greedy tree builder and has worst-case asymptotic time complexity $O(m + n \log n)$.

Kruskal's Minimum Spanning Tree Algorithm: This algorithm is a greedy minimum spanning tree algorithm that is different from the greedy tree builder. It has worst-case asymptotic time complexity $O(m \log n)$ where the dominant cost is sorting the edges based on their weights. After the edges are sorted, the remaining computation takes $O(m\alpha(m, n))$ time where α is the inverse Ackermann function (page 80).

Bellman-Ford Single-Source Shortest Path Algorithm: This algorithm computes a shortest path tree for a weighted graph that can have negative weight edges. It can also detect a negative weight cycle that is reachable from the source. This algorithm has $O(nm)$ worst-case asymptotic time complexity. Observe that when there are no negative weight edges, Dijkstra's algorithm is more efficient.

Floyd-Warshall All-Pairs Shortest Path Algorithm: A shortest path matrix is a set of n shortest path trees, one for each vertex as the source. This algorithm computes a shortest path matrix for a weighted graph that can have negative weight edges, and it can detect a negative weight cycle in the graph. This algorithm has $O(n^3)$ time complexity. When there are no negative weight edges, running Dijkstra's algorithm n times can compute the shortest path matrix in $O(mn + n^2 \log n)$ time, which is more efficient than the Floyd-Warshall algorithm.

Edmonds-Karp Maximum Flow Algorithm: This maximum flow algorithm uses the basic Ford-Fulkerson augmenting path methodology (page 948) using breadth-first search to find the augmenting path. It has worst-case asymptotic time complexity $O(n^2m)$. See [Section 57.9](#) for a discussion of more efficient maximum flow algorithms. By the **maximum-flow minimum-cut theorem**, a maximum flow in a graph also provides a minimum cut.

Uses: These weighted graph algorithms use many of the collection implementations including:

DynamicArray (Chapter 13) is used to sort the edges in the graph using quicksort as part of the preprocessing for Kruskal's minimum spanning tree algorithm.

OpenAddressing (Chapter 22) is used to return the set of edges in the minimum spanning tree computed by Prim's algorithm and by Kruskal's algorithm.

OpenAddressingMapping (Section 49.7.2) is used by Kruskal's algorithm for the mapping from each vertex to its union-find node.

TaggedPairingHeap (Section 49.8.4) is used by the greedy tree builder, which in turn is used by both Dijkstra's shortest path algorithm and Prim's minimum spanning tree algorithm.

Constructors

The constructor takes *rep*, the graph representation to use, and *multigraph*, which indicates if the graph is a multigraph.

```
public AbstractWeightedGraph(GraphRepresentation<V, E> rep, boolean multigraph) {
    super(rep, multigraph);
}
```

57.1 Greedy Tree Builder

In this section, we present a greedy algorithm to construct a tree for a weighted graph. In general a **greedy algorithm** is an algorithm in which locally optimal choices are made in an attempt to reach a globally optimal solution. While often greedy algorithms do not produce an optimal solution, there are several algorithms based on the greedy tree builder that do yield provable optimal solutions. In particular, we instantiate the greedy tree builder to obtain Dijkstra's shortest path algorithm (Section 57.2) and Prim's minimum spanning tree algorithm (Section 57.3). These two algorithms, and many others, share a common structure that we capture in the general-purpose **greedy tree builder** algorithm. The greedy tree builder has the following general structure. Some vertex *s* (that we call the *source*) is used as a seed to start building the tree *T*. All vertices reachable from *s* must eventually be added to *T*. Based on the specific application, a cost function is defined for each vertex $v \notin T$ that defines a cost for placing *v* into the tree based on the edge $e = (u, v)$ (or $\{u, v\}$ in an undirected graph) for some $u \in T$. The method *getCost* is defined as part of the *InTree* class (Section 53.3). This method takes as a parameter an edge *e* where $e.source \in T$ and $e.dest \notin T$, and the cost that was associated with *e.source* (the parent) when it was placed in *T*. It returns the cost of using *e* to connect *e.dest* to *T*. For clarity, we give three specific examples here.

Single-Source Shortest Path Problem: For this problem, T is a shortest path tree and s is the source vertex. The desired semantics for the cost of vertex $v \in T$ is that $\text{cost}(v)$ is the weight of the shortest path from s to v . The source is given a cost of 0. When considering adding v to T by using the edge $e = (u, v)$ where $u \in T$, the cost for v is $\text{cost}(u) + e.\text{weight}$ since that would be the weight of the path to v obtained using the path to u stored in T followed by e . The greedy choice made at each step is to select the vertex $v \notin T$ with the minimum cost to add to T . In Section 57.2 we prove that when a vertex v is added to T , the path from s to v in T is a shortest path. (Note that the paths are actually stored as an in-tree by the greedy tree builder.)

Minimum Spanning Tree Problem: For this problem, T is a spanning tree and s is an arbitrary seed vertex. The desired semantics for the cost of vertex $v \in T$ is that $\text{cost}(v)$ is the weight of the edge used to connect v to T . (Think of this as the cost of adding some infrastructure represented by the undirected edge $\{u, v\}$ to connect v to T .) Again s is given a cost of 0. When considering adding v to T using $e = \{u, v\}$, the cost for v is $e.\text{weight}$ since that is the cost of the edge connecting v to the tree. The greedy choice made at each step is to select the vertex $v \notin T$ with the minimum cost to add to T . In Section 57.3 we prove that this locally optimal decision (using the least cost edge to connect a new vertex to the spanning tree), yields a globally optimal solution (i.e., a minimum spanning tree of the graph).

Maximum Bottleneck Problem: For two specified vertices s and t , we define the **bottleneck** $b(P)$ of a path P in a directed weighted graph to be the minimum weight edge on the path. That is $b(P) = \min_{e \in P} w(e)$. The **maximum bottleneck path problem** is to find a path from s to t with a maximum bottleneck. The desired semantics for the cost of vertex $v \in T$ is that $\text{cost}(v)$ is the bottleneck in the maximum bottleneck path from s to v . The source s is given a cost of ∞ . When considering adding v to T using the edge $e = (u, v)$ where $u \in T$, the cost for v is $\min(u_{\text{cost}}, w)$ since that would be the bottleneck of the path to v obtained using the maximum bottleneck path already computed for u followed by e . Since the task is to find a maximum bottleneck path, the greedy choice made at each step is to select the vertex $v \notin T$ with the maximum cost to add to T .

Each of the above problems can be solved using the greedy tree builder. At each point in its execution, the greedy tree builder can be viewed as classifying each vertex u in one of three sets with the following semantics:

T (tree): The vertices in the greedy tree. A final decision has been made to place u into the greedy tree T . The parent from u does not change after u is placed in T .

Q (queue): The vertices in the queue that are discovered but not yet placed in T . At each step in the algorithm, an element from Q is selected to be added to T . At all times, each element in Q stores a reference to the parent it would have if it were added to T next. As the algorithm runs, these temporary references are updated over time to reflect the best connection currently known for each element. Vertices in Q are still seeking a better connection to the tree. Each application must define a priority for each vertex $v \in Q$ so that the “best connection” has the highest priority. For example, for the shortest path and the minimum spanning tree problems, the priority is defined to be inversely proportional to the cost. This choice ensures that the highest priority vertex is the one with the minimum cost. For the maximum bottleneck problem, the priority is defined to be proportional to the cost. This choice ensures that the highest priority vertex is the one with the highest cost.

U (undiscovered vertices): The vertices that have not yet been visited by the greedy tree builder.

Initially s is placed in T . Then the following steps are repeated until all discovered vertices have been placed in T .

1. Select the vertex $u \in Q$ with the highest priority over all vertices in Q . (For each algorithm, a proof that this greedy choice is part of an optimal solution is required to prove that the final solution is optimal.)
2. Remove u from Q , which implicitly places u in T . Since the cost for each vertex $v \in Q$ represents its best connection to some vertex in T , the addition of u to T provides a new possible connection for each vertex $v \notin T$.
3. Consider all outgoing edges $e = (u, v)$ from u .
 - (a) If $v \in U$, then v is placed into Q after setting the edge from its parent to e and initializing v_{cost} to the cost associated for parent edge e .
 - (b) If $v \in Q$, the cost associated with v , for parent edge e , is computed. If this cost c is better than v 's current cost, then the cost for v is set to c and its parent edge is set to e .

Note that the greedy tree builder adds vertices to the in-tree when it adds them to the queue. When we refer to T , we mean the tree including only those vertices in the in-tree that are not also in the queue. We say that an edge $e = (u, v)$ where $u \in T$ and $v \notin T$ is a **fringe edge** of T . Similarly, we say that v is a **fringe vertex**. When $e = (u, v)$ is a fringe edge of T , we say that the tree that would be obtained by adding e to T is an **extension** of T .

To efficiently implement these basic steps we need a fast way to find the highest priority fringe vertex, and to increase the priority of a fringe vertex when a better connection to T is found. Therefore, a priority queue is a good fit, since it provides an efficient way to find the highest priority item and to increase the priority of an item. Since the priority of a vertex is not an attribute of the vertex, we use a tracked tagged priority queue (Section 49.8). The reason to use a tracked version is important – it gives us a way to quickly change the priority of a vertex without incurring the linear time cost of searching for the vertex. The comparator to use for the tagged priority queue and the cost to be associated with the source are provided as parameters to the greedy tree builder.

Greedy Tree Builder Data (GTBD) Inner Class

Recall that the `TreeData` instance associated with vertex v in an in-tree stores the cost and parent edge of v . For the greedy tree builder, the tracker for the tagged priority queue must also be maintained for each vertex so that the vertex can be efficiently located in the priority queue when a connection for it is found. In addition, a method *consider* is provided to consider a new connection for v to the greedy tree, and to update the tree and priority queue when this connection is a better one.

We extend `TreeData` to create the abstract greedy tree builder data (GTBD) class that is used by greedy tree builder. The single instance variable *loc*, a locator for v in the tagged priority queue, is added.

```
static abstract class GTBD<V, E> extends TreeData<V, E> {
    PriorityQueueLocator<TaggedElement<Double, V>> loc;

    GTBD(E edge, double cost) {
        super(edge, cost);
    }
}
```

The *getCost* function takes e , the edge leading to v from its parent and *parentCost*, the cost associated with the parent. It returns the cost v would have if e were the edge from its parent.

Each application for the greedy tree builder must provide an implementation for this method. For example, for Dijkstra's shortest path algorithm the cost is the length of the path from the source to the parent (*parentCost*) plus the weight of the edge *e* going from the parent to *v*.

abstract double *getCost*(E *e*, **double** *parentCost*);

Whenever the greedy tree builder removes the highest priority vertex *u* from the priority queue, each outgoing edges *e* = (*u*, *v*) from *u* must be considered to see if using edge *e* provides a better connection to the tree for *v*. The *consider* method takes *e*, the edge being considered, and *parentCost*, the cost associated with *u*, the new parent candidate. If replacing the current edge from the parent for *u* to *e* increases the priority for *v*, then this change is made. Often, this method is called **relax**.

```
void consider(E e, double parentCost, TaggedPriorityQueue<Double, V> pq) {
    double newCost = getCost(e, parentCost);
    if (newCost < loc.get().getTag()) {
        edgeFromParent = e;
        cost = newCost;
        pq.updateTag(cost, loc);
    }
}
```

Greedy Tree Builder Method

The *greedyTreeBuilder* method takes *tree*, an (initially empty) in-tree to be constructed, *seedCost*, the cost that should be associated with the first vertex placed in the tree, and *comp*, the comparator to apply to the cost function so that the highest priority vertex is the one that should be placed in the tree next. This method throws a *NegativeWeightEdgeException* when the algorithm cannot guarantee a correct result due to a negative weight edge.

The only initialization performed before the main loop is to insert the source (or seed) vertex into the tree and into the tagged priority queue with value *seedCost*. While the priority queue is not empty, the vertex *u* with the highest priority is removed. The inner while loop iterates over all outgoing edges *e* from *u*. If the other vertex *v* of *e* is not in the tree, then it is added to the tree and to priority queue with the cost returned by *getCost*. If *v* is already in the tree, then *consider* is used to see if edge *e* can provide a lower cost for *v*, and if so *e* becomes the parent edge for *v*.

For the heap methods that are used by the greedy tree builder, the heap data structure with the best known asymptotic time complexity is the Fibonacci heap. However, the experiments of Moret and Shapiro [116] show that pairing heap is empirically better, so we have selected a tagged pairing heap for our implementation.

```
public void greedyTreeBuilder(InTree<V, E> tree, double seedCost,
    Comparator<Double> comp) throws NegativeWeightEdgeException {
    TaggedPriorityQueue<Double, V> pq = new TaggedPairingHeap<Double, V>(comp);
    GTBD<V, E> vData = (GTBD<V, E>) tree.add(tree.source, null, seedCost);
    vData.loc = pq.putTracked(seedCost, tree.source);
    while (!pq.isEmpty()) {
        V u = pq.extractMax().getElement();
        GTBD<V, E> uData = (GTBD<V, E>) tree.get(u);
        Iterator<E> outEdges = edgesFrom(u);
        while (outEdges.hasNext()) {
```

```

    E e = outEdges.next();
    if (e.weight() < 0)
        throw new NegativeWeightEdgeException(e.toString());
    V v = neighbor(u, e);
    if (tree.contains(v)) {
        vData = (GTBD<V, E>) tree.get(v);
        if (vData.loc.inCollection()) //if v is still in the priority queue
            vData.consider(e, uData.cost, pq); //consider if e gives better connection
    }
    else { //just discovered v, so put into tree and queue
        vData = (GTBD<V, E>) tree.add(v, e, vData.getCost(e, uData.cost));
        vData.loc = pq.putTracked(vData.cost, v);
    }
}
}
}

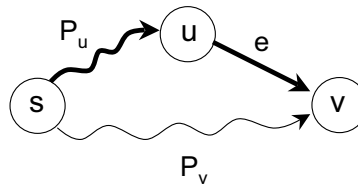
```

The correctness argument for this algorithm is dependent on the particular application, so for each application a correctness proof is needed. However, the time complexity analysis is independent of the application as long as the *getCost* method takes constant time. We analyze the time complexity of *greedyTreeBuilder* under this assumption. For the tagged priority queue *pq*, we assume that *add* takes $O(T_a(n))$, *extractMax* takes $O(T_e(n))$ time, and *updateTag* to increase the priority takes $O(T_u(n))$ time where n is the number of elements in the priority queue.

The initialization takes expected constant time since the source is added into an empty priority queue, and placed into the tree (an open addressing mapping). When a vertex is removed from the priority queue it is placed in the tree, and never removed from the tree. So each vertex is placed in the priority queue at most once. Thus, there are at most n executions of the while loop, and *extractMax* is executed at most n times. Also, the outgoing edges for each vertex are traversed at most once. Thus, like in the analysis of breadth-first search and depth-first search, during the entire execution of this algorithm, the outgoing edges from each vertex reachable from s are traversed exactly once. For an adjacency list representation the total cost is $O(n + m)$, and for an adjacency matrix representation the total cost is $O(n^2 + m)$.

We now consider the computation that occurs for each edge e that is considered in the inner while loop. In the worst case, this computation is performed m times, once for each edge in the graph.

- The weight of e is checked to see if it is a negative weight edge. This takes constant time.
- The neighbor v of e is obtained, which takes constant time.
- The *contains* method is used to see if v is in T . This takes expected constant time since the tree is an open addressing mapping from vertices to GTBD instances.
- The priority queue locator *inCollection* method is used to determine if v remains in the priority queue. This step takes constant time.
- When v is in the priority queue, the *consider* method is used to replace v 's connection into T if u provides a better connection. This step involves increasing the priority of the element in the priority queue, which has $O(T_u(n))$ cost, the dominant cost of the algorithm.
- When v is newly discovered, it is added to the tree, which has expected constant cost, and added to the priority queue, which has $O(T_a(n))$ cost. Observe that this step can be executed

**Figure 57.1**

An illustration of edge relaxation using the edge e to find a path better than P_v from s to v . A wavy line illustrates a path that can be composed of one or more edges in the original graph. If the weight of the path P_u plus the weight of e is less than the weight of P_v , then a shorter path to v has been discovered.

at most once for each vertex, since a vertex is only placed into the priority queue once.

Thus, the worst-case cost for the greedy tree builder when using an adjacency list representation for the graph is

$$O(nT_a(n) + nT_e(n) + mT_u(n) + n + m) = O(n(T_a(n) + T_e(n)) + mT_u(n)),$$

and when using an adjacency matrix representation for the graph, the worst-case cost is

$$O(nT_a(n) + nT_e(n) + mT_u(n) + n^2 + m) = O(n^2 + n(T_a(n) + T_e(n)) + mT_u(n)).$$

The asymptotically fastest implementation is obtained using the tagged Fibonacci heap (Section 49.8.5) which has amortized costs of $T_a(n) = O(1)$, $T_e(n) = O(\log n)$, and $T_u(n) = O(1)$. This leads a *worst-case* cost for the greedy tree builder of $O(n \log n + m)$ when using an adjacency list, and a cost of $O(n^2 + m)$ when using an adjacency matrix. When a pairing heap is used, the greedy tree builder has time complexity $O(m \log n)$ when using an adjacency list, and $O(n^2 + m \log n)$ when using an adjacency matrix. Since G could be a multigraph, it is possible that $m = \omega(n^2)$.

57.2 Dijkstra's Single-Source Shortest Path Algorithm

Dijkstra's algorithm solves the single-source shortest path problem for a weighted graph in which all weights are non-negative. The goal is to find a shortest path from a specified source vertex s to all other vertices, where the length of path p is the sum of the edge weights along the path.

Recall that breadth-first search (Section 53.4) solves the single-source shortest path problem in an unweighted graph. At each step, breadth-first search selects the vertex u at the front of the queue of discovered vertices to add to the shortest path tree. All the outgoing edges from u are examined, and any newly discovered vertices are added to the end of the queue. The crux of the correctness argument for breadth-first search relies on the fact that for the path from vertices in s to u in the shortest path tree, plus the edge from u to v is a shortest path from s to v in the graph.

Breadth-first search discovers the vertices in non-decreasing order based on the cost of the shortest path from the source. This property allows a queue to be used to maintain the order in which to process the vertices, so that when edge $e = (u, v)$ is considered, if v is already in the queue, then e cannot provide a better connection for v into the shortest path tree. For a weighted graph, neither of these properties hold, so a more complex algorithm is required.

The difficulties described in the last paragraph are overcome by replacing the queue by a priority queue, and updating the priority for v when a shorter path from s is discovered via the edge (u, v)

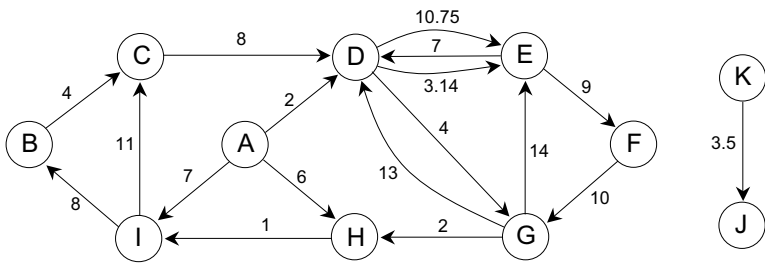


Figure 57.2
A weighted directed multigraph.

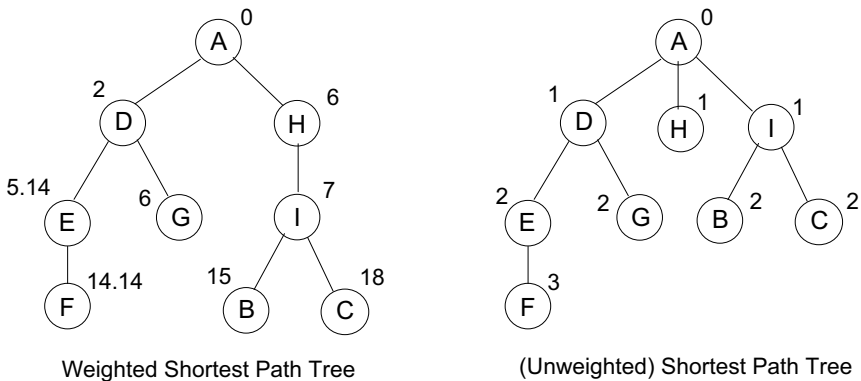


Figure 57.3
The weighted shortest path tree (as computed by Dijkstra’s algorithm), and the unweighted shortest path tree (as computed by breadth-first search) for the graph of Figure 57.2.

for $u \in T$. Dijkstra's algorithm is amenable to a greedy tree builder implementation in which the three sets of vertices have the following semantics:

T (**tree**): the set of vertices for which the shortest path has been computed,

Q (**queue**): the set of vertices v still in the priority queue, meaning that the path from s to v in the in-tree might not be a shortest path, and

U (**undiscovered**): the set of vertices that are not yet discovered.

Note that the greedy tree builder adds vertices to the in-tree when it adds them to the queue. When we refer to T , we mean the tree including only those vertices in the in-tree that are not also in the priority queue.

Initially $T = \{s\}$, where s is the source and all other vertices are in U . For each vertex $v \in Q$, the following key invariant is maintained: the path stored for v in the in-tree is a shortest path from s to v when no vertex in Q , besides v , is on the path. A vertex is in U when it is not reachable from any vertex in T . We use v_{cost} to denote the cost of the current path in the in-tree from s to v for each $v \in T \cup Q$. That is, $v_{\text{cost}} = \text{tree.get}(v).\text{cost}$.

Dijkstra's shortest path algorithm repeats the following steps until Q is empty.

1. Select the vertex $v \in Q$ that minimizes v_{cost} over all vertices in Q . Since there are no negative weight edges, the path in the in-tree to u must be a shortest path, since there are no vertices in $Q \cup U$ that are closer to s than u .
2. Remove u from Q , which implicitly places u in T .
3. Consider all outgoing edges $e = (u, v)$ from u . Observe that a possible path from s to v can be obtained by using the shortest path P_u from s to u followed by the edge from u to v . The cost of the resulting path is $u_{\text{cost}} + e.\text{weight}()$. Let P_v be the shortest known path from s to v . (See Figure 57.1.)
 - (a) If $v \in U$ (P_v does not exist), then v is placed into Q after setting the edge from its parent to e and initializing v_{cost} to $u_{\text{cost}} + e.\text{weight}()$.
 - (b) If $v \in Q$ and $u_{\text{cost}} + e.\text{weight}() < w(P_v)$, P_v is a better path. We **accept** P_v as the shortest path found so far from s to v by setting v 's parent edge to e in the in-tree, and updating the cost for v to $u_{\text{cost}} + e.\text{weight}()$. This process is often known as **edge relaxation** using e , and is performed in our implementation by the *consider* method of the greedy tree builder data (GTBD) class.

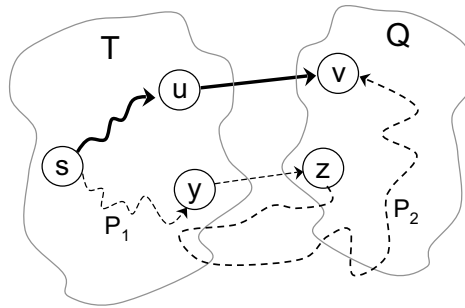
Figure 57.3 shows the shortest path tree obtained by Dijkstra's algorithm on the weighted graph of Figure 57.2. The unweighted shortest path tree as computed by breadth-first search is also shown. We use the greedy tree builder (Section 57.1) to implement Dijkstra's shortest path algorithm.

Weighted Shortest Path Data (WSPD) Inner Class

The greedy tree builder cost function is defined in the weighted shortest path data (WSPD) class that extends the greedy tree builder data (GTBD) class. For Dijkstra's algorithm when considering edge $e = (u, v)$ as a way to reach vertex v from vertex u , the cost for vertex v would be $u_{\text{cost}} + e.\text{weight}$.

```
static class WSPD<V, E> extends GTBD<V, E> {
    WSPD(E edge, double cost) { super(edge, cost); }
```

Recall that *getCost* takes e , an edge leading to this vertex from a new possible parent vertex u and *parentCost*, the cost associated with u . It returns the cost for this vertex when reached by way of edge e from u .

**Figure 57.4**

An illustration for the inductive step in the proof correctness for Dijkstra's single-source shortest path algorithm. The path P is shown as a bold edge, and the path P^* is shown as a dashed path.

```

double getCost(E e, double parentCost) {
    return parentCost + ((WeightedEdge) e).weight();
}

```

Implementation for Dijkstra's Algorithm

The *weightedShortestPaths* method takes s , the source vertex, and uses the greedy tree builder to build a shortest path tree with the specified source. It throws a *NegativeWeightEdgeException* when the graph has a negative weight edge reachable from s . The comparator used for the greedy tree builder is defined so that the highest priority vertex is the one with the lowest cost. The cost given to the source is 0 since an empty path (which by definition has cost 0) is used to reach s .

```

public InTree<V, E> weightedShortestPaths(V s) throws NegativeWeightEdgeException {
    InTree<V, E> tree = new InTree<V, E>(s) {
        GTBD<V, E> add(V v, E edgeFromParent, double cost) {
            GTBD<V, E> data = new WSPD<V, E>(edgeFromParent, cost);
            put(v, data);
            return data;
        }
    };
    greedyTreeBuilder(tree, 0.0, new Comparator<Double>() {
        public int compare(Double e1, Double e2) {
            return e2.compareTo(e1);
        }
    });
    return tree;
}

```

*//highest priority vertex
//is one with minimum value tag*

Correctness Highlights: We now prove that the greedy tree builder yields an optimal solution. That is, we prove that for each $v \in V$ the path in the in-tree returned from the greedy tree builder is a shortest path from s to v in the graph. We denote the length of the shortest path from s to v by $d(v)$. We state and prove a two part invariant to show that (1) vertices in T have the correct

cost, and (2) that the cost associated with each vertex $v \in Q$ is the cost of the minimum cost extension of T by v . More formally,

- $\forall v \in T, v_{\text{cost}} = d(v)$, and
- $\forall v \in Q, v_{\text{cost}}$ is the value of the minimum cost extension of T by v .

Base: Initially $T = \emptyset$, and $Q = \{s\}$ for which $s_{\text{cost}} = d(s) = 0$. So both invariants initially hold.

Inductive step: Assume that when $|T| = i$, both invariants hold. We now prove that the invariants are preserved when the next vertex is added to T . Let v be the $(i + 1)^{\text{st}}$ vertex extracted from the priority queue, and hence implicitly moved from Q to T , and let P be a path from s to v with weight v_{cost} , which clearly exists and is defined by following the sequence of parent edges. Let u be the vertex just before v in P . Paths to vertices in Q are only considered if they are extensions of T , so $u \in T$ and by the inductive hypothesis $u_{\text{cost}} = d(u)$.

We now prove that P is a shortest path from s to v . Suppose not. Then there exists a path P^* from s to v where $w(P^*) = d(v) < w(P)$. Since P^* connects a vertex in T to a vertex in $V - T$, there must be a first fringe edge $(y, z) \in P^*$ where $y \in T$ and $z \in V - T$. We can partition the path P^* as $P_1 + (y, z) + P_2$ where P_1 is the portion of P^* from s to y and P_2 is the portion of P^* from z to v and “+” is concatenation. (See Figure 57.4.) By the inductive hypothesis $y_{\text{cost}} = d(y)$. Furthermore since P^* is a shortest path, it follows that $P_1 + (y, z)$ must be a shortest path from s to z . A shorter path from s to z would contradict the optimality of P^* . By the second invariant, $\text{cost}(z)$ is the cost of a minimum extension of T by z . Combined, these two facts imply that $z_{\text{cost}} = d(z)$. Since both v and z were in the priority queue when v was removed, it follows that $v_{\text{cost}} \leq z_{\text{cost}}$. Furthermore, since the edge weights are all non-negative $d(v) = w(P^*) \geq z_{\text{cost}}$. This combined with $v_{\text{cost}} \leq z_{\text{cost}}$ yields that $w(P^*) \geq v_{\text{cost}} = w(P)$ which contradicts that $w(P^*) < w(P)$, completing the inductive proof for the first invariant.

When v is placed in T , all outedges (v, w) of v are considered, and if a better path to w is obtained by appending (v, w) to the shortest path for v , then w_{cost} and $w.\text{edgeFromParent}$ are updated accordingly. Thus, the second invariant is preserved.

Upon completion of the greedy tree builder, Q is empty and the shortest paths in T are optimal. Finally, since all the outgoing edges from the vertices in T are examined, any vertices that remain in U are unreachable. ■

The time complexity for Dijkstra’s algorithm is that of the greedy tree builder. When using an adjacency list for the graph representation, and a tagged Fibonacci heap (Section 49.8.5) for the tagged priority queue, it has worst-case $O(n \log n + m)$ time complexity.

57.3 Prim’s Minimum Spanning Tree Algorithm

Given an undirected weighted graph $G = (V, E)$, in which there are no negative edges, a **minimum spanning tree** is an acyclic subset of edges $E' \subseteq E$ for which $G = (V, E')$ is a single component and $\sum_{e \in E'} e.\text{weight}$ is minimized. Prim’s minimum spanning tree algorithm also uses the greedy tree builder. For Prim’s algorithm, the three sets of vertices used by the greedy tree builder have the following semantics:

T (**tree**): the set of vertices that have been placed in the final spanning tree,

Q (**queue**): the set of vertices that are in the priority queue, and

U (**undiscovered**): the set of vertices that are not yet discovered.

Suppose T has fringe edge $e = \{u, v\}$ with $u \in T$ and $v \notin T$. For Prim's minimum spanning tree algorithm, when considering edge e as an extension of T , v_{cost} should correspond to the cost of adding v , which is $e.\text{weight}$. As in Dijkstra's algorithm, the priority of a vertex is inversely related to the cost. The resulting algorithm works as follows. An arbitrary vertex s is selected as the starting vertex, which forms a singleton tree. When s is removed from the priority queue and added to T , all vertices reachable from it are placed in the priority queue with a cost equal to the weight of the edge to s . In general, the cost associated with a vertex v is the least cost edge from v to some vertex in T . At each step, the vertex u with the least cost connection to some vertex in T is moved from Q to T . Then all vertices v incident to u are considered. If the weight of the edge e between u and v is less than the current cost for vertex v , then the cost for vertex v is updated to be the weight of e , and e is saved as the *edgeFromParent* for v . Unlike in Dijkstra's algorithm where the graph is directed, the edges in the spanning tree are undirected. The *edgeFromParent* for vertex v corresponds to the edge that is placed in the spanning tree when v is removed from the priority queue.

Figure 57.5 shows an undirected weighted graph, and the spanning tree computed by Prim's minimum spanning tree algorithm. The vertex s is the "seed" for the spanning tree. While the choice for s is arbitrary, when there is more than one minimum spanning tree, the choice for s can affect which minimum spanning tree is computed. Regardless of the choice for s , the sum of the edge weights in the minimum spanning tree is the same. When the edges incident to s are considered, c is placed in the priority queue with a tag of 3, g is placed in the priority queue with a tag of 8, and h is placed in the priority queue with a tag of 5. Since c has the highest priority (smallest) tag, it is removed from Q and placed into T (using the edge $\{s, c\}$). As a result, b (with a tag of 5), d (with a tag of 8), and f (with a tag of 5) are added to Q .

Next one of the three vertices with a tag of 5 would be removed from Q . For this example, suppose that b is removed and added to T (using the edge $\{b, c\}$). The edge $\{b, h\}$ is considered, but it does not provide a better connection for h . When the edge $\{b, a\}$ is considered, it provides a connection for a , so a is placed in Q with a tag of 10.

Next, one of the two vertices in Q with a tag of 5 would be removed from Q . Suppose that h is removed and added to T (using the edge $\{s, h\}$). The edge $\{g, h\}$ provides a less expensive connection for vertex g to T , so the priority for g is increased by changing its tag to 1. The edge $\{h, a\}$ is considered, but it does not provide a better connection for a . Next g is added to T (using the edge $\{g, h\}$), which will cause the tag for f to be updated to 2, and so on. The resulting spanning tree is shown as bold edges in Figure 57.5.

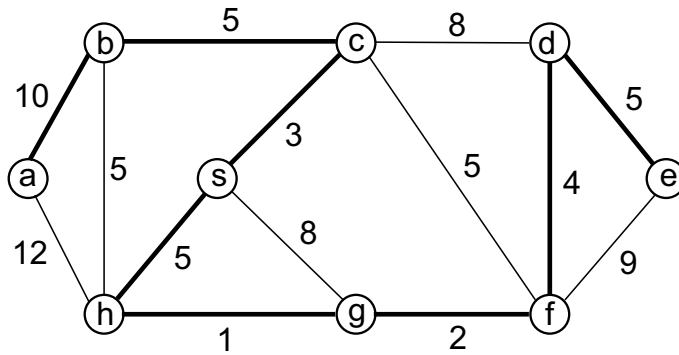
Minimum Spanning Tree Data (MSTD) Inner Class

The greedy tree builder cost function is defined in the minimum spanning tree data (MSTD) class that extends the greedy tree builder data (GTBD) class.

```
static class MSTD<V, E> extends GTBD<V, E> {
    MSTD(E edge, double cost) {
        super(edge, cost);
    }
}
```

Recall that the *getCost* function takes e , the edge connecting this vertex with its parent and *parentCost*, the cost associated with the parent. It returns the cost associated with the edge. For Prim's minimum spanning tree algorithm, the cost is just the edge weight. There is no dependence on the cost of the parent.

```
double getCost(E e, double parentCost) {
    return ((WeightedEdge) e).weight();
}
}
```

**Figure 57.5**

A weighted undirected graph with the minimum spanning tree computed by Prim's algorithm shown as bold edges. The vertices are added to the spanning tree in the order s (the seed), c (using $\{s, c\}$), b (using $\{b, c\}$), h (using $\{s, h\}$), g (using $\{h, g\}$), f (using $\{g, f\}$), d (using $\{f, d\}$), e (using $\{d, e\}$), a (using $\{b, a\}$). Although we could show the spanning tree as a rooted tree, as in the shortest path tree, the edges of the spanning tree are undirected.

Implementation for Prim's Algorithm

The *primMST* method returns a set of edges that is a minimum spanning tree for this graph. It throws a *DisconnectedGraphException* when the graph is not connected, throws a *NegativeWeightEdgeException* when a negative edge weight is encountered, and throws a *GraphException* when the graph is directed.

As with Dijkstra's shortest path algorithm, the comparator used for the greedy tree builder is defined so that the highest priority vertex is the one with the lowest cost. The cost given to the source is 0, the $n - 1$ edges in the spanning tree correspond to the edges used to connect the remaining $n - 1$ vertices to T .

```

public Set<E> primMST()
    throws NegativeWeightEdgeException, DisconnectedGraphException {
    if (isDirected())
        throw new GraphException("undirected graphs expected");
    V seed = iterator().next(); //use an arbitrary vertex as the seed
    InTree<V, E> mstTree = new InTree<V, E>(seed) {
        GTBD<V, E> add(V v, E edgeFromParent, double cost) {
            GTBD<V, E> data = new MSTD<V, E>(edgeFromParent, cost);
            put(v, data);
            return data;
        }
    };
    greedyTreeBuilder(mstTree, 0.0, new Comparator<Double>() {
        public int compare(Double e1, Double e2) {
            return e2.compareTo(e1);
        }
    });
    Set<E> mst = new OpenAddressing<E>(numVertices()-1); //create set to return
    for (V v: this)
        if (v != seed)
            try {

```

```

        mst.add(mstTree.get(v).edgeFromParent);
    } catch (NoSuchElementException nsee) {
        throw new DisconnectedGraphException();
    }
}
return mst;
}

```

Correctness Highlights: We argue inductively that when vertex v is added to T , the set of edges in T are a subset of some minimum spanning tree. Initially just the seed vertex s is in T without any edges, so the invariant clearly holds. Let $V_T = \{V_1, V_2, \dots, V_k\}$ be the connected components (defined on page 847) of $G' = (V, T)$. For the inductive step we focus on the **compacted graph** G_T for the undirected graph $G = (V, E)$ with respect to T , where $G_T = (V_T, E_T)$ where

$$E_T = \{\{V_i, V_j\} \mid (u, v) \in E \text{ and } u \in V_i \text{ and } v \in V_j\}.$$

Observe that G_T can be a multigraph, even when G is not. If desired, only a minimum weight edge between V_i and V_j need be kept in G_T .

By the inductive hypothesis, T is a subset of a minimum spanning tree of G . Thus, the problem of computing a minimum spanning tree in G is equivalent to computing a minimum spanning tree in G_T . Let V_s be the vertex in V_T that contains s . In Prim's minimum spanning tree algorithm, every other vertex in V_T is a single vertex in the original graph. To prove the inductive step, we show that the next edge added to the spanning tree for the input G_T is part of a minimum spanning tree of G_T . By construction of the greedy tree builder, the edge $e = \{v_i, V_s\}$ selected is a minimum-weight edge incident to V_s . Suppose that there is no minimum spanning tree of G_T that includes edge e . Let T^* be a minimum spanning tree of G_T . Since it is a spanning tree, there must be a path P in T^* between v_i and V_s . Let e' be the maximum weight edge in P , and define $T_e^* = T^* \cup \{e\} - \{e'\}$. First we argue that T_e^* is a spanning tree in G_T . Observe that adding e to T^* creates a cycle C . When removing e' all vertices in C are still reachable, so T_e^* is a spanning tree. (For example, in [Figure 57.5](#) adding $\{c, f\}$ to the spanning tree shown by the thick edges, creates the cycle $\langle c, f, g, s, c \rangle$. If $\{s, h\}$ is removed, then what remains is a different minimum spanning tree.) Finally, since e is a minimum weight edge incident to s , $w(T_e^*) \leq w(T^*)$. Thus T_e^* is a minimum spanning tree of G_T that contains e , contradicting the assumption that there is not one. This completes the proof the inductive hypothesis. ■

The time complexity for Prim's algorithm is that of the greedy tree builder. When using an adjacency list for the graph representation, and a tagged Fibonacci heap (Section 49.8.5) for the tagged priority queue, it has worst-case $O(n \log n + m)$ time complexity.

57.4 Kruskal's Minimum Spanning Tree Algorithm

Kruskal's minimum spanning tree algorithm is a greedy algorithm to compute a minimum spanning tree. However, it does not use the greedy tree builder technique. Instead, Kruskal's algorithm first sorts all edges by weight in non-decreasing order and initializes the spanning tree T to be empty. The edges in E are considered in sorted order (i.e., from smallest to largest). If $T \cup \{e\}$ is acyclic, then e is added to T . Therefore, as the algorithm runs, T is a forest until finally all of the trees are connected to form a spanning tree of the entire graph. This is a greedy algorithm since at each step

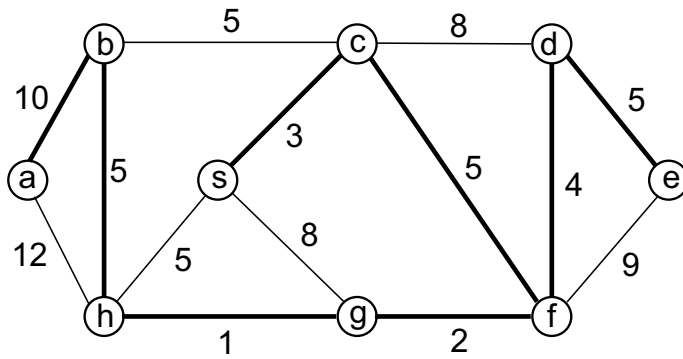


Figure 57.6

A weighted undirected graph with the minimum spanning tree computed by Kruskal's algorithm. The edges are added in the order $\{g, h\}$, $\{f, g\}$, $\{s, c\}$, $\{d, f\}$, $\{d, e\}$, $\{c, f\}$, $\{a, b\}$. The edges with weight 5 could be added in any order. Also $\{s, h\}$ could replace $\{c, f\}$, and $\{b, c\}$ could replace $\{c, f\}$.

a greedy decision is made to add the minimum weight edge to the spanning tree T that does not create a cycle in T . Figure 57.6 shows the minimum spanning tree created by Kruskal's algorithm using the same graph as in Figure 57.5.

In the implementation of Kruskal's algorithm, a data structure is needed to efficiently determine if adding an edge would create a cycle in the graph. If T were a tree throughout the algorithm, this could be done easily by checking if both endpoints were already in T . However, since T is built up as multiple components, an edge connecting two vertices in T is permissible as long as the two vertices are in different components. Depth-first search could be used to detect cycles, but at each step only one additional edge is added to the graph, so a much more efficient solution can be obtained using the union-find data structure (Chapter 6). In particular, there will be one union-find node for each component. Initially each vertex forms its own component. If the edge being considered is between two vertices in the same component, then it would form a cycle, and should not be added to the spanning tree. Otherwise, the edge is added to the spanning tree, and the two components that it connects are merged. A mapping from each vertex to its associated union-find node is used to efficiently locate a vertex in the union-find data structure.

The *kruskalMST* method returns a set of edges that is a minimum spanning tree for this graph. It throws a *DisconnectedGraphException* when the graph is not connected, throws a *NegativeWeightEdgeException* when a negative edge weight is encountered, and throws a *GraphException* when the graph is directed.

```

public Set<E> kruskalMST()
    throws NegativeWeightEdgeException, DisconnectedGraphException {
    if (isDirected())
        throw new GraphException("undirected graphs expected");
    DynamicArray<E> edgeList = new DynamicArray<E>();
    for (V v: graph) {
        Iterator<E> edges = edgesFrom(v);
        while (edges.hasNext()) {
            E e = edges.next();
            if (e.weight() < 0)
                throw new NegativeWeightEdgeException(e.toString());
            edgeList.add(e);
        }
    }
}

```



```

edgeList.quickSort(new Comparator<E>() {
    public int compare(E e1, E e2) { //compare edges according to weight
        return Double.compare(e1.weight(), e2.weight());
    }
});
Mapping<V, UnionFindNode<V>> vertexMap =
    new OpenAddressingMapping<V, UnionFindNode<V>>(numVertices());
for (V v: graph)
    vertexMap.put(v, new UnionFindNode<V>(v));
Set<E> mst = new OpenAddressing<E>(numVertices()-1);
for (E edge : edgeList){
    UnionFindNode<V> u = vertexMap.get(edge.source());
    UnionFindNode<V> v = vertexMap.get(edge.dest());
    if (u.findRepresentative() != v.findRepresentative()) {
        mst.add(edge);
        u.union(v);
        if (mst.getSize() == numVertices()-1)
            return mst;
    }
}
throw new DisconnectedGraphException();
}

```

Correctness Highlights: The correctness argument is very similar to that for Prim's algorithm. The only difference is that each of the vertices in G_T may be composed of several vertices from the original graph, and the edge added is the minimum weight edge among all edges that go between distinct vertices in G_T , as opposed to the minimum weight edge incident to V_s . The rest of the proof is the identical. ■

The expected time complexity to sort the edges is $O(m \log m) = O(m \log n)$ when there are at most $O(n^k)$ edges for some constant k . Since the minimum weight edge between u and v is the only one that would be used, when G is a multi-graph an $O(m)$ preprocessing step could be used to remove from the edge list all but a least cost edge connecting each pair of vertices. In our implementation, we have selected quicksort (Section 11.4.5). If desired, mergesort (Section 11.4.2) could instead be used to give a worst-case $O(m \log n)$ cost to sort the edges.

We now consider the cost for Kruskal's algorithm after the edges are sorted. It takes constant time (per edge) to iterate through the list of sorted edges, and takes expected constant time to get the source and destination vertex from the open addressing mapping. It also takes expected constant time to add an edge to mst . Finally, there are at most 2 calls to *findRepresentative* and at most 1 call to *union* per edge. As discussed in our presentation of the union-find data structure (Section 6.3), Tarjan [144, 145] proved that if there is a sequence of calls to *union* and *findRepresentative*, when the sequence includes y calls that simply call the constructor to create a new partition element, then the worst-case time complexity is $O(x\alpha(x, y))$. In this application of the union-find data structure, there are at most $3m$ method calls made to the union-find data structure, where n calls add a new element. Thus, after the sorting is completed, the remainder of Kruskal's algorithm has worst-case $O(m\alpha(m, n))$ complexity, which for all practical purposes is linear in the number of edges.

57.5 Bellman-Ford's Single-Source Shortest Path Algorithm

We now generalize the single-source shortest path problem to allow negative weight edges. Observe that if there is a path from the source vertex s to a vertex in a negative weight cycle, then the shortest path from s to any other vertex is not well defined because the weight of the path that traverses the cycle infinitely many times approaches $-\infty$. So a single-source shortest path algorithm should either find a negative weight cycle reachable from the source, or otherwise find a shortest path from s to all vertices in the graph reachable from s .

Like the greedy tree builder, the Bellman-Ford algorithm is based on the notion of **edge relaxation**. (See Figure 57.1.) In particular, suppose that the shortest path known so far from source s to vertex u is P_u with weight $w(P_u)$. Likewise, suppose that the shortest path known so far from the source s to vertex v is P_v with weight $w(P_v)$. Let e be an edge from u to v with weight w_e . If $w(P_u) + w_e < w(P_v)$ then the path obtained by appending e to the end of P_u is a better path to v than P_v . When edge e is considered in Dijkstra's shortest path problem, it is known that P_u is a shortest path from s to u . Thus, the edge from u to v need be considered only once. However, when there are negative weight edges, such a guarantee is not possible. For example, it could be that a path of length 15 is known for v , a path of length 12 is known for u and for $e = (u, v)$, $w_e = 5$. When e is considered under these conditions, it does not provide a better path to v . However, the path of length 12 for u might not be shortest path. Suppose that a relaxation on an edge with source u replaces the shortest path for u with one of weight 3. When e is considered under these conditions, it provides a better path to v .

Bellman-Ford's algorithm is very simple. A **relaxation pass** iterates through all edges in the graph, and performs edge relaxation for each edge. The Bellman-Ford algorithm performs $n - 1$ relaxation passes. The intuition behind the correctness is as follows. After k passes over the edges, a shortest path will have been found for each vertex for whose shortest path from s has at most k edges. If there are no negative weight cycles reachable from s , then it follows that the shortest path from s to any other vertex can use at most $n - 1$ edges. There is no benefit in repeating an intermediate vertex since that would define a (non-negative weight) cycle in the graph that could only increase the weight of the path. To see why only $n - 1$ passes are needed, consider a hypothetical n^{th} pass over the edges. If this pass would result in a better path being found, then the edge e used in the relaxation must be part of a negative weight cycle in the graph. As illustrated in depth-first search (Section 53.5), the shortest path tree can then be used to construct a negative weight cycle that involves e .

The *generalShortestPathFromSource* method takes *source*, the source vertex. It returns a shortest path tree for the provided source. This method implements the Bellman-Ford single-source shortest path algorithm.

```
public InTree<V, E> generalShortestPathFromSource(V source) {
    InTree<V, E> tree = new InTree<V, E>(source);
    for (V v: this) { //initialization
        if (v == source) //place the source in the shortest path tree with cost 0
            tree.add(source, null, 0);
        else //place all other vertices in the shortest path tree with infinite cost
            tree.add(v, null, INF);
    }
    Set<E> edges = edgeSet(); //set of edges in the graph to iterate over
    for (int i = 1; i < numVertices(); i++) { //repeat n-1 times
        for (E e : edges) //consider edge e as a way to improve path from source to dest
```

```

    TreeData<V, E> uData = tree.get(e.source());
    TreeData<V, E> vData = tree.get(e.dest());
    if (vData.cost > uData.cost + e.weight()) {           //if e provides a better path to dest
        vData.cost = uData.cost + e.weight();             //update the cost for dest
        vData.edgeFromParent = e;                         //and update parent edge for dest
    }
}
}
//could stop here if it is known that there are no negative weight cycles
for (E e : edges){                                       //check for a negative weight cycle using one more iteration
    if (tree.get(e.dest()).cost > tree.get(e.source()).cost + e.weight())
        tree.cycleStart = e.dest();                     //record the fact that a cycle exists involving e.dest
}
return tree;
}

```

Correctness Highlights: First we argue that when the graph has no negative weight cycle reachable from the source s , the in-tree returned contains a shortest path from s to every reachable vertex v .

We prove inductively that after the i^{th} relaxation pass, for any vertex v in which the shortest path from s to v uses at most i edges, the shortest path tree T contains a shortest path for v . For $i = 0$ this invariant holds since only s has a length 0 shortest path to itself, and it is initialized with a distance of 0 (and an empty path). To prove the inductive step, suppose that all vertices in which the shortest path consists of at most k edges is correctly computed in T at the end of the k^{th} relaxation pass. Let vertex v be a vertex in which the shortest path from s to v has $k + 1$ edges. Let u immediately precede v in a shortest path from s to v . Since the path from s to u has only k edges, by the inductive hypothesis the path for s in T is optimal. Let e be the edge from u to v . This edge is considered at some point during the $(k + 1)^{\text{st}}$ relaxation pass. When e is considered, the optimal cost for a path from s to v is compared to the cost of the current path. Either the parent edge for v is updated to e , or a different optimal path has already been found. In either case, at the end of the $(k + 1)^{\text{st}}$ relaxation pass, the path for v in T is a shortest path.

We now prove that when there is no negative weight cycle reachable from s , the shortest path from s to any other vertex v has at most $n - 1$ edges. Suppose not. Let P be a shortest path from s to v that uses the fewest possible edges. By assumption, P must have at least n edges, which implies that $n + 1$ vertices are used by P . So some vertex of the graph appears twice in P , which implies that P has some cycle C . If C is a negative weight cycle, this contradicts the assumption there is no negative weight cycle. So C must have no weight. However, removing C from P yields an optimal path with fewer edges than P , contradicting that P has the fewest edges of a shortest path. Thus P can have at most $n - 1$ edges. This completes the correctness proof when the graph has no negative weight cycles.

Finally, we consider when there is a negative weight cycle reachable from s . Let the cycle be $\langle v_0, \dots, v_{k-1}, v_k \rangle$ where $v_k = v_0$, and let e_i connect v_i to v_{i+1} . Let d_i be the length of the shortest path found so far for v_i . We now prove that for some edge e_i in this cycle, $d_i + w(e_i) < d_{i+1}$, which implies that a vertex in the negative weight cycle is properly recorded. Suppose not. Then

$$d_0 + w(e_0) \geq d_1$$

$$d_1 + w(e_1) \geq d_2$$

$$d_2 + w(e_2) \geq d_3$$

$$\begin{aligned} & \vdots \\ d_{k-2} + w(e_{k-2}) & \geq d_{k-1} \\ d_{k-1} + w(e_{k-1}) & \geq d_0 \end{aligned}$$

Summing both sides yields $(d_0 + \cdots + d_{k-1}) + (w(e_0) + \cdots + w(e_{k-1})) \geq d_0 + \cdots + d_{k-1}$, which implies that $w(e_0) + \cdots + w(e_{k-1}) \geq 0$, contradicting that C is a negative weight cycle. ■

We now analyze the time complexity analysis of the Bellman-Ford algorithm. The initialization takes expected $O(n)$ time since n insertions are made into an open addressing mapping. When using an adjacency list as the graph representation, the *edgeSet* method takes $O(n + m)$ time, and when using an adjacency matrix as the graph representation, the *edgeSet* method takes $O(n^2)$ time. The inner for loop, which performs a relaxation step on an edge takes constant time. Since each edge is considered n times, the overall worst-case asymptotic time complexity is $O(nm)$.

57.6 Shortest Path Matrix

package graph

```
TaggedCollectionWrapper<V, InTree<V,E>> implements TaggedCollection<V, InTree<V,E>>
└─ OpenAddressingMapping<V, InTree<V,E>> implements Mapping<V, InTree<V,E>>
    └─ ShortestPathMatrix<V, E extends Edge<V>>
        implements Mapping<V, InTree<V,E>>
```

So far, we have considered finding the shortest paths between a *single source* and all other vertices in the graph. The problem of finding a shortest path between *all pairs* of vertices in a graph is known as the **all-pairs shortest path problem**. If a graph has no negative weight edges, then the most efficient solution for this problem is to run Dijkstra's algorithm for each $u \in V$ to obtain the shortest path tree T_u for u as the source. The time complexity for this solution is $O(mn + n^2 \log n)$. When the graph has negative weight edges, the Bellman-Ford single-source shortest path algorithm could be applied once for each vertex, yielding an $O(n^2m)$ algorithm. In the next section, we present the Floyd-Warshall all-pairs shortest path algorithm that provides a more efficient $O(n^3)$ solution.

In this section, we lay the groundwork for that algorithm by defining the *ShortestPathMatrix* class that is used to store the result from an all-pairs shortest path algorithm, just as the *InTree* class holds the result from a single-source shortest path algorithm. The standard representation for a shortest path matrix is an $n \times n$ array, where the vertices have indices $\{0, 1, \dots, n - 1\}$ and where array entry (i, j) stores a shortest path (represented via the parent edge) from v_i to v_j . Our shortest path matrix is instead represented as a mapping from u to T_u , a shortest path tree for source vertex u . The advantage of this approach is that the public methods to obtain a shortest path from u to v can just delegate this operation to the shortest path tree for vertex u . Also, the space usage of maintaining a mapping of mappings is comparable to that required to store a two-dimensional array along with the mapping that would be required from the vertices to their indices in $\{0, 1, \dots, n - 1\}$. Note that while our shortest path matrix is stored as n shortest path trees, the Floyd-Warshall algorithm

simultaneously computes all n shortest path trees, so the shortest path matrix should be viewed as a single object.

Instance Variables and Constants: The instance variable *cycleStart* references a vertex that is known to be part of a negative weight cycle, or is *null* if there is no negative weight cycle.

V cycleStart = **null**;

Shortest Path Matrix Methods

The constructor takes g , the weighted graph on which the shortest path matrix is defined. The for loop initializes the shortest path matrix so that the shortest path tree for vertex u has

- $d(u, u) = 0$, a weight of zero for u itself (since a path of length 0 can be used), and
- $d(u, v) = \infty$ for $v \neq u$.

Then for each outgoing edge e from u , if $e.weight$ is less than $d(u, e.dest)$, then the data associated with u for vertex $v = e.dest$ is updated to correspond to edge e . Thus, upon completion of the initialization $d(u, v)$ is the value of the minimum weight edge from u to v where there is an implicit edge of weight 0 from each vertex to itself.

```
ShortestPathMatrix(WeightedGraph<V, E> g) {
    for (V u: g) {                                     //initialization
        InTree<V, E> uTree = new InTree<V, E>(u);
        put(u, uTree);
        for (V v: g)                                   //start with
            if (u == v)                                //(u, v) entry of 0 for u = v
                uTree.add(v, null, 0);
            else                                         //and (u, v) entry of infinity u != v
                uTree.add(v, null, AbstractGraph.INF);
        Iterator<E> outEdges = g.edgesFrom(u);
        while (outEdges.hasNext()) {                   //consider all edges from u
            E e = outEdges.next();                     //next edge e
            V v = e.dest();                             //is to vertex v
            TreeData<V, E> uvData = uTree.get(v);
            //if e is cheaper (u, v) edge, then replace edge stored in uvData by e
            if (uvData.cost == AbstractGraph.INF ||
                e.weight() < uvData.edgeFromParent.weight()) {
                uvData.edgeFromParent = e;
                uvData.cost = e.weight();
            }
        }
    }
}
```

The *hasNegativeWeightCycle* method returns *true* if and only if the graph has a negative weight cycle.

```
public boolean hasNegativeWeightCycle() {
    return cycleStart != null;
}
```

The *getNegativeWeightCycle* method returns a positional collection that holds the edges in a negative weight cycle, or *null* if there is no negative weight cycle. This method uses *buildCycleStartingAt* from the *InTree* class.

```
public PositionalCollection<E> getNegativeWeightCycle() {
    if (hasNegativeWeightCycle())
        return get(cycleStart).buildCycleStartingAt(cycleStart);
    return null;
}
```

The *getInTreeForSource* method takes *source*, the desired source vertex. It returns a shortest path tree for the given source vertex.

```
public InTree<V, E> getInTreeForSource(V source) {
    return get(source);
}
```

Finally, we include two methods for convenience to directly get a shortest path or the cost of a shortest path between a specified pair of vertices. The functionality of these methods could be achieved by using the *getInTreeForSource*, and then the appropriate *InTree* method.

The *getPath* method takes *source*, the source vertex, and *dest*, the destination vertex. It returns a positional collection that holds the edges, in order, in a shortest path from *source* to *dest*. If there is no path from *source* to *dest* then *null* is returned.

```
public PositionalCollection<E> getPath(V source, V dest) {
    return get(source).getPathFromSource(dest);
}
```

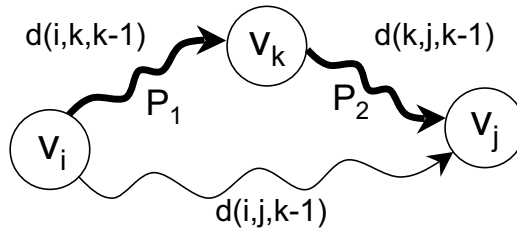
The *getPathDistance* method takes *source*, the source vertex, and *dest*, the destination vertex. It returns the length of the shortest path from *source* to *dest*, or *INF* (infinity) if there is no path from *source* to *dest*.

```
public double getPathDistance(V source, V dest){
    return get(source).get(dest).cost;
}
```

57.7 Floyd-Warshall's All-Pairs Shortest Path Algorithm

In this section, we present the Floyd-Warshall all-pairs shortest path algorithm. Let $G = (V, E)$ be a weighted directed graph, possibly with negative weight edges. Let $V = \{v_1, \dots, v_n\}$. As proven in Section 57.5, if a graph has no negative weight cycles then the shortest path between any two vertices is a simple path (i.e., there is no cycle). We use w_{ij} to denote the weight of the edge (if any) from v_i to v_j , where $w_{ij} = \infty$ if there is no edge in G from v_i to v_j . We use $d(i, j, k)$ to denote the length of a shortest path from v_i to v_j where the intermediate vertices on the path are selected from among $\{v_1, \dots, v_k\}$. We use the convention that when $k = 0$, no intermediate vertices are allowed. Consider the following recursive definition for $d(i, j, k)$:

$$d(i, j, k) = \begin{cases} 0 & \text{for } i = j \\ w_{ij} & \text{for } k = 0 \\ \min(d(i, j, k-1), d(i, k, k-1) + d(k, j, k-1)) & \text{otherwise} \end{cases}$$

**Figure 57.7**

An illustration of the recursive definition for $d(i, j, k)$. Observe that the best path from v_i to v_j when restricted to use intermediate vertices from the set $\{v_1, \dots, v_k\}$ either includes v_k as an intermediate vertex, or does not. The path shown by combining the two paths shown in bold, corresponds to the case when v_k is used in the shortest path from v_i to v_j . The other possibility is that the shortest path from v_i to v_j that uses intermediate vertices $\{v_1, \dots, v_k\}$ is the same as the shortest path from v_i to v_j that just uses intermediate vertices in $\{v_1, \dots, v_{k-1}\}$.

where w_{ij} denotes the minimum weight edge from v_i to v_j (or ∞ if there is no edge from v_i to v_j).

By definition $d(i, j, 0) = w_{ij}$. Intuitively, to compute $d(i, j, k)$, either the shortest path includes v_k , or it does not. (See Figure 57.7.) When v_k is not included then $d(i, j, k) = d(i, j, k-1)$, and when v_k is included then the weight of shortest path from i to j (through v_k) is defined by $d(i, k, k-1) + d(k, j, k-1)$. The correctness argument proves inductively that $d(i, j, k)$ is indeed the length of the shortest path from v_i to v_j with intermediate vertices in $\{v_1, \dots, v_k\}$. Finally, observe that $d(i, j, n)$ is the length of the shortest path in G from v_i to v_j since all possible intermediate vertices are allowed.

The Floyd-Warshall algorithm is based on this recursive definition for $d(i, j, k)$. At a high level, a three-dimensional matrix M can be allocated where entry i, j, k holds a data object that includes both $d(i, j, k)$ and the last edge on a shortest path from i to j using only intermediate vertices $\{v_1, \dots, v_k\}$. This edge serves exactly the same role as *parentEdge* does in the shortest path tree. While it would be easy to create an algorithm that directly implements the recursive definition, terminating when $k = 0$, the repetition of subproblems using a direct recursive solution leads to an exponential time solution. Bellman [20] introduced **dynamic programming** as a general technique to avoid solving the same subproblem from a recursive definition more than once, by iteratively computing M from the bottom of the recursion upward. More specifically, for the Floyd-Warshall algorithm, the portion of M for which $k = 0$ corresponds to the original edge weights. Once M is computed for a given value of k , then in $O(n^2)$ all entries for the next value of k are each computed in constant time. This leads to an $O(n^3)$ algorithm. One important observation, which can reduce the space complexity of the Floyd-Warshall algorithm, is that all entries of M need not be stored. Instead, a single two-dimensional matrix can be used that holds the shortest path from i to j using the current value of k . Furthermore, observe that since $d(i, k, k-1) = d(i, k, k)$ and $d(k, j, k-1) = d(k, j, k)$, the minimization in the recursive definition can be viewed as the relaxation step used in the other shortest path algorithms. If $d(i, k) + d(k, j) < d(i, j)$ then the new value and corresponding parent edge can replace the current values.

The *allPairsShortestPaths* algorithm returns a shortest path matrix. The first portion of this method computes the shortest paths. The outer for loop introduces a new intermediate vertex *mid* (v_k in the discussion above). Then, for every pair of vertices, *source* (i in the discussion above) and *dest* (j in the discussion above), it compares the length of the current best path known from *source* to *dest* to the one obtained by concatenating the best known path from *source* to *mid* with the best known path from *mid* to *dest*. As in the general relaxation step, if a better path is found, then both the cost of the path and the *edgeFromParent* are stored.

As in the Bellman-Floyd algorithm, a final loop is used to check for a negative weight cycle in the graph. Since the resulting matrix contains the shortest path between each pair of vertices (including a vertex to itself), it can be shown that there is a negative weight cycle in the graph that includes vertex v exactly when the shortest path from v to itself is negative. The first such vertex, if any, is recorded in *cycleStart*. If *cycleStart* is *null* after this method terminates, there is no negative weight cycle in the graph.

```

public ShortestPathMatrix<V, E> allPairsShortestPaths() {
    ShortestPathMatrix<V, E> shortestPaths = new ShortestPathMatrix<V, E>(this);
    for (V mid: this) { //mid is being considered as an intermediate vertex
        InTree<V, E> midT = shortestPaths.get(mid);
        for (V source: this) {
            InTree<V, E> sourceT = shortestPaths.get(source);
            for (V dest: this){
                double potentialCost = sourceT.get(mid).cost + midT.get(dest).cost;
                if (potentialCost < sourceT.get(dest).cost) { //found better path
                    sourceT.get(dest).cost = potentialCost;
                    sourceT.get(dest).edgeFromParent = midT.get(dest).edgeFromParent;
                }
            }
        }
    }
    shortestPaths.cycleStart = null; //no negative weight cycle detected yet
    for (V v: this) { //check for a negative weight cycle in the graph
        if (shortestPaths.get(v).get(v).cost < 0) {
            shortestPaths.cycleStart = v; //detected a negative weight cycle at v
            break;
        }
    }
    return shortestPaths;
}

```

Correctness Highlights: First we consider when G has no negative weight cycle. We use induction on k to prove that the recursive definition for $d(i, j, k)$ is correct. By definition $d(i, j, 0) = w_{ij}$. We now prove that for $k \geq 1$, if $d(i, j, k-1)$ is correctly computed for all i, j , then $d(i, j, k)$ is correctly computed for all i, j . Let P be a shortest path from v_i to v_j when restricted to use intermediate vertices from the set $\{v_1, \dots, v_k\}$. Either P includes v_k as an intermediate vertex, or it does not. If v_k is not in P , then $d(i, j, k) = d(i, j, k-1)$, and by the inductive hypothesis $d(i, j, k-1)$ is correctly computed. We now consider when v_k is in P . Let P_1 be the portion of P from v_i to v_k and let P_2 be the portion of P from v_k to v_j . (See Figure 57.7.) We now argue that P_1 must be a shortest path from v_i to v_j . Suppose not. Let P'_1 be a shorter path from v_i to v_j . Then P'_1 followed by P_2 yields a shorter path than P from v_i to v_j , contradicting that P is a shortest path. Likewise, P_2 must be a shortest path from v_k to v_j . Also, since the graph has no negative weight cycles, P must be a simple path. Thus v_k cannot be in P_1 , and v_k cannot be in P_2 . Thus by the inductive hypothesis, $d(i, k, k-1)$ is the length of P_1 , and $d(k, j, k-1)$ is the length of P_2 . Thus, $d(i, j, k) = d(i, k, k-1) + d(k, j, k-1)$. Since the better of these two options (P includes v_k as an intermediate vertex or it does not) is selected, it follows that $d(i, j, k)$ is correctly computed for all i, j .

Since the shortest path is always a simple path, the shortest path between any two vertices that involves v_k need not include v_k as an intermediate vertex. It follows that $d(i, k, k-1) =$

$d(i, k, k)$ and $d(k, j, k - 1) = d(k, j, k)$. Thus, replacing the original recursive definition by:

$$d(i, j, k) = \begin{cases} 0 & \text{for } i = j \\ w_{ij} & \text{for } k = 0 \\ \min(d(i, j, k - 1), d(i, k, k) + d(k, j, k)) & \text{otherwise} \end{cases}$$

still yields a correct solution. So, as long as $d(i, j, 0)$ is computed for all i, j , and then $d(i, j, 1)$ is computed for all i, j , and so on, $d(i, j, n - 1)$ will hold a shortest simple path between all pairs of vertices. If the graph does not have any negative weight cycle, then these paths are shortest paths.

Finally, if the graph has a negative weight cycle that includes vertex v , then the shortest simple path from v to v would be negative. So the negative weight cycle is detected in the final loop. ■

We now analyze the time complexity for the Floyd-Warshall algorithm. The initialization takes $O(n^2)$ time. The dominant cost is to compute the shortest path for all i, j as each of the vertices is introduced as a new candidate for the intermediate vertex. This portion takes $O(n^3)$ time since its three nested for loops each iterate over the n vertices. Finally, the check for a negative weight cycle takes $O(n)$ time.

57.8 Edmonds-Karp Maximum Flow Algorithm

In this section we present an algorithm for finding a maximum flow (or equivalently a minimum cut) in a flow network. Recall that a **flow network** is a weighted directed graph in which all edge weights are non-negative. Without loss of generality, we assume that the graph is not a multigraph since a set of edges from u to v could be replaced by a single edge whose weight that is the sum of the individual edge weights. We assume that the weight of edge e corresponds to the capacity of e , the amount of flow that can be routed along e . A flow network specifies a source vertex $s \in V$ that produces the commodity, and a sink (destination) vertex $t \in V$ that consumes the commodity. A flow is a function $f : E \rightarrow \mathbb{R}$ that indicates how much flow is carried by each edge.

As defined, a flow network has a single-source (the producer), and a single sink (the consumer). However, for many applications there may be multiple producers and multiple consumers. The following standard reduction allows a flow problem with input $G = (V, E)$ with $P \subseteq V$ the set of producers and $C \subseteq V$ the set of consumers, to be reduced to a maximum flow problem with input $G' = (V', E')$ where G' has a single producer s and a single consumer t . In this reduction, G' is only slightly larger than G so there is little increase in the time complexity caused by this transformation. Let $V' = V \cup \{s, t\}$ where s (the “super” source) and t (the “super” sink) are two new vertices that were not in V . The edge set E' will contain all edges in E along with infinite weight edges $\{(s, v) \mid v \in P\}$ and $\{(v, t) \mid v \in C\}$. That is, there is an infinite capacity edge from s to each producer in G , and infinite capacity edge from each consumer in G to t . It is easily seen that a maximum flow for G' includes a maximum flow for G by using the flow just on the edges in E .

We let f_{uv} denote $f(e)$, where $e = (u, v)$. If there is no such edge then $f_{uv} = 0$. A flow f is a **legal** flow if it satisfies the following constraints*. The first says that an edge cannot carry more flow than its capacity, and the second says that the total flow into an intermediate vertex must equal the total flow out of that vertex.

*Some presentations use a convention called **skew symmetry** that $f_{uv} = -f_{vu}$. We do not use this convention.

Capacity Constraint: $f(e) \leq e.\text{weight}$

Flow Conservation: For all $v \in V - \{s, t\}$,
$$\left(\sum_{e \in v.\text{inedges}} f(e) \right) = \left(\sum_{e \in v.\text{outedges}} f(e) \right)$$

Recall that when $f(e) = e.\text{weight}$, we say that e is **saturated**. The value for flow f is

$$|f| = \left(\sum_{e \in s.\text{outedges}} f(e) \right),$$

the total flow out of the source. A legal flow f is a **maximum flow**, when for any other legal flow f' , $|f| \geq |f'|$. Such a flow corresponds to the maximum amount of the commodity, per unit time, that can be transported from s to t in the flow network.

Recall that a **cut** of a weighted graph $G = (V, E)$ is a partition of the vertices into (V_1, V_2) . That is, $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \emptyset$. Let $E_{(V_1, V_2)} = \{e = (u, v) \in E \mid u \in V_1 \text{ and } v \in V_2\}$. The

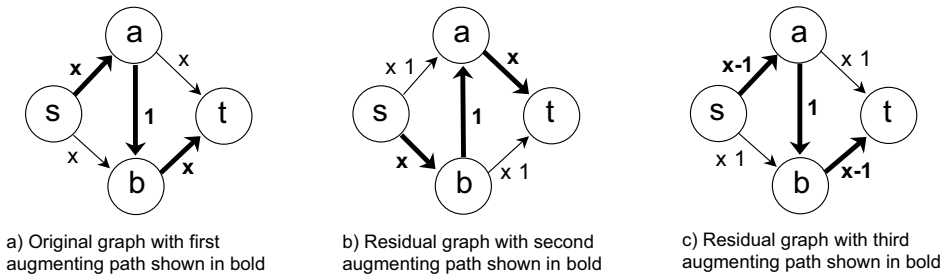
capacity of cut $C = (V_1, V_2)$ in a directed graph $G = (V, E)$ is $w(C) = \left(\sum_{e \in E_{(V_1, V_2)}} e.\text{weight} \right)$.

An (S, T) -cut is a cut for which $s \in S$, $t \in T$. A minimum cut of a flow network is an (S, T) cut C such that for any other (S, T) cut C' , $w(C) \leq w(C')$.

We now describe the basic approach introduced by Ford and Fulkerson [58] to compute a maximum flow in a flow network $G = (V, E)$ with a single source s and a single sink t . Given G and a legal flow f , the **residual graph** $G_f = (V, E_f)$ is defined as follows. Each non-saturated edge $e \in E$ is placed into E_f with a weight of $e.\text{weight} - f(e)$. That is, each non-saturated edge in G is placed in G_f with the capacity equal to the residual capacity that remains on that edge. In addition, for each edge $e = (u, v) \in E$ for which $f(e) > 0$, the edge (v, u) is placed in E_f with capacity $f(e)$. These edges allow any portion of flow that was routed along e to be returned so that it can be routed a different way. An **augmenting path** is a path from s to t in the residual graph.

The basic Ford-Fulkerson method begins with a flow f that assigns 0 to each edge. It then repeatedly finds an augmenting path P from s to t and adds the minimum edge weight in P to the flow for all edges in P . As discussed later in this section, when there is no path from s to t in the residual graph, then f is a maximum flow. While the basic Ford-Fulkerson method is guaranteed to yield an optimal solution, it may not be efficient. In particular, it is possible that only a single unit of flow is added during each augmentation. Figure 57.8 illustrates a simple input when $|f|$ augmentations may occur, leading to $\Omega(|f|)$ time complexity. In the flow network of Figure 57.8, x can be made arbitrarily large, and $|f| = 2x$ for the maximum flow.

Independently, Dinic [47] and Edmonds and Karp [50] proved that polynomial-time performance could be guaranteed by using as the augmenting path an unweighted shortest path (measured by the number of edges) in the residual graph from s to t . In particular, the Edmonds-Karp maximum flow algorithm uses breadth-first search (Section 53.4) to find the augmenting path. Observe that in the example of Figure 57.8, after augmenting along the $\langle s, a, t \rangle$ and $\langle s, b, t \rangle$ paths (in either order), a maximum flow is computed. Figure 57.9 illustrates a sample execution of the Edmonds-Karp algorithm on a different graph. The top left graph (a) is the original graph, which is also the initial residual graph. Breadth-first search in the residual graph yields the path shown in bold in graph (a). The minimum edge weight on this residual path is 2, so 2 units of flow are added to all edges in the path. The resulting flow graph is shown at the top right (b), where edges with no flow are omitted. The residual graph that corresponds to this flow is shown in the bottom left (c). Observe that the residual graph includes an edge from f to d , which does not occur in the original graph. A shortest unweighted path from s to t is shown in bold, with a minimum edge weight of 1. Therefore, observe

**Figure 57.8**

An example to illustrate that the time complexity of the basic Ford-Fulkerson method can be arbitrarily large. Observe that the edge weight x can be any value.

that one unit of flow that was going from d to f is rerouted. The resulting flow, which is a maximum flow, is shown at the bottom right (d).

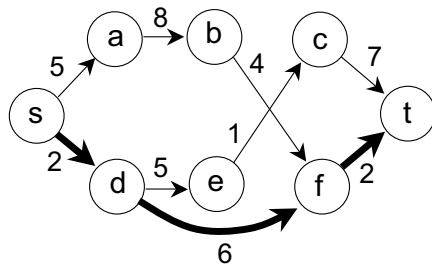
We now describe our implementation of the Edmonds-Karp algorithm. We construct a flow graph that is returned, after removing any edges with zero flow. Initially, a zero weight edge is placed in the flow graph corresponding to each edge $e \in E$. If edge $(u, v) \in E$ and $(v, u) \notin E$, then (v, u) is also added to the flow graph. Observe that the residual graph changes with each augmentation step. To save both the space and time required to build the residual graph, we instead create a “virtual” residual graph by providing the method *weightInResidualGraph* that uses the flow graph and original graph to determine the weight in the residual graph. The residual graph is never explicitly built.

Flow Graph Inner Class

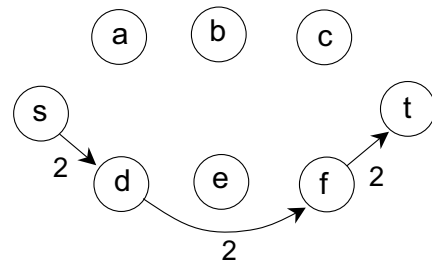
The *FlowGraph* class extends *WeightedAdjacencyMatrix*. The class *SimpleWeightedEdge* (Section 56.5), which just includes a source, destination, and weight, is used for the edges.

The constructor first adds all vertices from V into the flow graph. Then for each edge (u, v) in the original graph, the edge (u, v) and (v, u) are added to the flow graph with weight 0. The method *removeZeroWeightEdges* is called after execution of the maximum flow algorithm to remove from the flow graph all edges that still have zero weight.

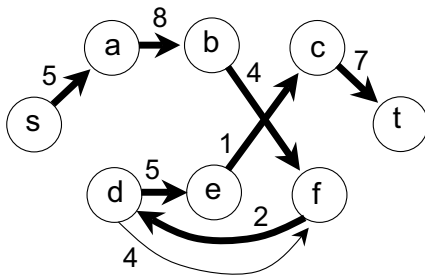
```
public class FlowGraph extends WeightedAdjacencyMatrix<V, SimpleWeightedEdge<V>> {
    FlowGraph() {
        for (V v: AbstractWeightedGraph.this)
            addVertex(v);
        for (E e: AbstractWeightedGraph.this.edgeSet()) {
            if (!containsEdge(e.source(), e.dest()))
                addEdge(new SimpleWeightedEdge<V>(e.source(), e.dest(), 0));
            if (!containsEdge(e.dest(), e.source()))
                addEdge(new SimpleWeightedEdge<V>(e.dest(), e.source(), 0));
        }
    }
    void removeZeroWeightEdges() {
        for (SimpleWeightedEdge<V> e: edgeSet()) {
            if (e.weight() == 0)
                removeEdge(e);
        }
    }
}
```



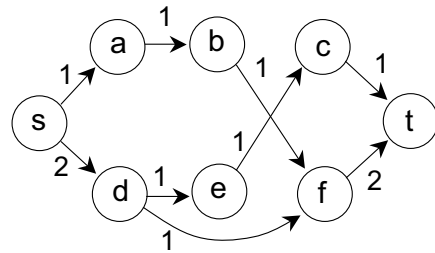
a) Original graph with first augmenting path shown in bold



b) Flow after first augmentation, edges with no flow are not shown



c) Residual graph based on flow shown in graph (b) with the augmenting path shown in bold



d) Maximum flow obtained after second augmentation

Figure 57.9

The execution of the Ford/Fulkerson method on a small flow network. We use the Edmonds-Karp technique of using breadth-first search to find an augmenting path from s to t with the fewest edges possible.

Implementation for the Edmonds-Karp Algorithm

The *weightInResidualGraph* method takes *src*, the source vertex, *dest*, the destination vertex, and *flow*, the flow graph. It returns the weight of the edge from *src* to *dest* in the residual graph defined by *flow*.

```
double weightInResidualGraph(V src, V dest, FlowGraph flow) {
    E graphEdge = getEdge(src, dest);
    SimpleWeightedEdge<V> reverseFlow = flow.getEdge(dest, src);
    double residualWeight = 0;
    if (graphEdge != null) //unused forward capacity
        residualWeight += graphEdge.weight() - flow.getEdge(src, dest).weight();
    if (reverseFlow != null) //used backward capacity available to push back
        residualWeight += reverseFlow.weight();
    return residualWeight;
}
```

The *findAugmentingPath* method takes s , the source vertex, t , the sink vertex, and *flow*, the flow graph. This method uses a procedure like breadth-first search to find a shortest path in the residual

graph from s to t . The breadth-first search stops once t is discovered since there is no need to continue the search. As an optimization, the same `InTree` object could be reused for each call to `findAugmentingPath` instead of creating a new one each time.

```

InTree<V, SimpleWeightedEdge<V>> findAugmentingPath(V s, V t, FlowGraph flow) {
    InTree<V, SimpleWeightedEdge<V>> residualFlowTree =
        new InTree<V, SimpleWeightedEdge<V>>(s);
    residualFlowTree.add(s, null, 0); //source is the root for breadth-first search
    Queue<V> q = new Queue<V>(); //create an empty queue of vertices
    q.enqueue(s); //place source in the queue
    while (!q.isEmpty()) { //while some vertex is still in the queue
        V u = q.dequeue(); //u is next vertex in the queue
        Iterator<SimpleWeightedEdge<V>> outEdges = flow.edgesFrom(u);
        while (outEdges.hasNext()) {
            SimpleWeightedEdge<V> e = outEdges.next(); //e is edge from u to v
            V v = e.dest(); //v is reachable from u by edge e
            if (u != v && (weightInResidualGraph(u, v, flow) > 0)) {
                if (!residualFlowTree.contains(v)) { //v is newly discovered
                    residualFlowTree.add(v, e, 0);
                    if (v == t)
                        return residualFlowTree;
                    q.enqueue(v); //put v at the end of the queue
                }
            }
        }
    }
    return null; //There is no augmenting path
}

```

The `augmentFlow` method takes `flow`, the flow graph, `s`, the source vertex, `t`, the sink vertex, and `residualFlowTree`, the shortest path tree from executing breadth-first search on the residual graph defined by `flow`. This method first traverses the path from t to s in the residual flow tree, to find the minimum weight edge on the path from s to t . The instance variable `bottleneck` will hold the weight of this edge. The second while loop augments the path P in the residual graph from s to t by `bottleneck`. It computes `backflow`, the amount of flow going in the reverse direction of the edge in P , and `pushback`, the amount of flow to reroute from the reverse edge. Any portion of the `bottleneck` flow that cannot be achieved by rerouting flow, is routed by using the edge (`flowEdge`) in P .

```

void augmentFlow(FlowGraph flow, V s, V t,
    InTree<V, SimpleWeightedEdge<V>> residualFlowTree) {
    double bottleneck = INF;
    V curVertex = t;
    while (curVertex != s) { //find minimum weight edge on path
        SimpleWeightedEdge<V> e = residualFlowTree.get(curVertex).edgeFromParent;
        bottleneck = Math.min(bottleneck, weightInResidualGraph(e.source(), curVertex, flow));
        curVertex = e.source();
    }
    curVertex = t; //now do the augmentation
    while (curVertex != s) {
        SimpleWeightedEdge<V> flowEdge = residualFlowTree.get(curVertex).edgeFromParent;
        SimpleWeightedEdge<V> reverseEdge = flow.getEdge(curVertex, flowEdge.source());
    }
}

```

```

    double backflow = reverseEdge.weight;
    double pushback = Math.min(backflow, bottleneck);
    reverseEdge.weight -= pushback;
    flowEdge.weight += (bottleneck - pushback);
    curVertex = flowEdge.source();
  }
}

```

The *maximumFlow* method takes *s*, the source vertex, and *t*, the sink vertex. It returns a flow graph with the maximum flow from *s* to *t*, for which only edges with non-zero flow are included. As long as there is a path in the residual graph from *s* to *t*, flow is augmented along this path. Since *findAugmentingPath* finds the augmenting path with breadth-first search, it finds a shortest unweighted path from *s* to *t* in the residual graph defined by the current flow. Thus, this implements the Edmonds-Karp maximum flow algorithm. When there is no augmenting path from *s* to *t*, all zero weight edges are removed from the flow graph and it is returned.

```

public FlowGraph maximumFlow(V s, V t) {
    FlowGraph flow = new FlowGraph();
    InTree<V, SimpleWeightedEdge<V>> residualFlow;
    while ((residualFlow = findAugmentingPath(s, t, flow)) != null)
        augmentFlow(flow, s, t, residualFlow);
    flow.removeZeroWeightEdges();
    return flow;
}

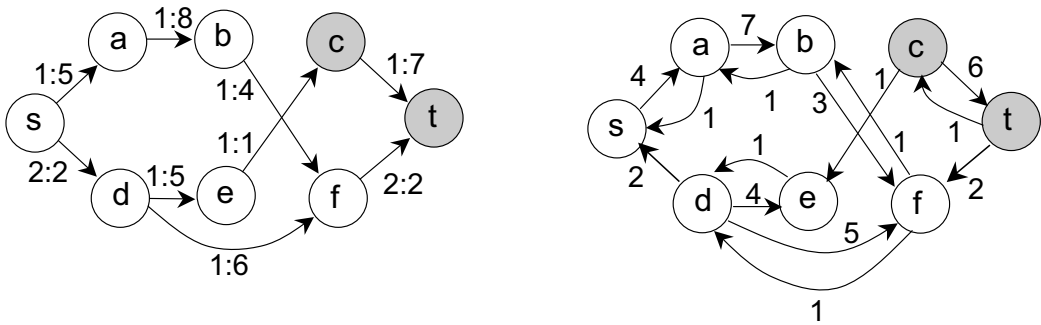
```

Correctness Highlights: We now prove that the flow graph returned contains a maximum flow from the source *s* to the sink *t*. We first observe that *flow* preserves both the capacity constraint and flow conservation. Initially the flow on all edges is zero, which satisfies both of these constraints. It is also easily seen that augmenting the flow by the minimum edge weight in the augmenting path preserves both constraints.

We now prove that upon termination, *flow* is a maximum flow. This portion of the proof relies upon the **max-flow min-cut theorem**, which states that when there is no path in the residual graph from the source to the sink, then the current flow must be a maximum flow. Figure 57.10 provides some intuition behind this theorem. The left side of the figure shows the original graph, and the flow on each edge. Also, observe that all edges between an unshaded and shaded vertex are saturated. The graph on the right is the residual graph. Observe that all unshaded vertices are reachable from *s*, but the unshaded vertices are not reachable. Intuitively, the shaded vertices are not reachable since the edges into them from any unshaded vertex are saturated. Thus, it is not possible to increase the flow from *s* to *t*. Furthermore, the total flow from *s* to *t* is equal to the capacity of some cut of *G*. That is, a maximum flow defines a minimum cut.

Suppose there is no augmenting path from *s* to *t* in the residual graph. Then the residual graph can be partitioned into two groups: *S*, the set of vertices reachable from the source, and *T* the set of vertices not reachable from the source. Thus, (*S*, *T*) is a cut *C* in the graph. Furthermore, any edge (*u*, *v*) in the original graph where *u* ∈ *S* and *v* ∈ *T* must be saturated, since otherwise there would be an edge from a vertex in *S* to a vertex in *T* in the residual graph, which would contradict that no vertex in *T* is reachable. All flow from *s* to *t* must go through *C*, so if all edges in *C* are saturated, then it is not possible to increase the flow from *s* to *t*. Therefore, *f* is a maximum flow.

We now prove that *C* is a minimum (*S*, *T*)-cut. Suppose there were some other cut *C'* where *w*(*C'*) < *w*(*C*). Since *f* is a valid flow where |*f*| = *w*(*C*), at least *w*(*C*) units of flow must go

**Figure 57.10**

On the left is the flow network from Figure 57.9 with the flow of an optimal solution shown for each edge. The shaded and unshaded vertices illustrate a minimum cut in the graph. The figure on the right is the residual graph. All unshaded vertices are reachable from s , and the shaded vertices are not reachable from s .

through any (S, T) cut, which contradicts that $w(C') < w(C)$. Thus, C is a minimum cut. ■

Finally, we overview the time complexity analysis. There are two key properties that are used in the analysis. The first property states that the number of edges on the augmenting path is non-decreasing during the execution of the algorithm. The second property states that any edge that is saturated during an augmentation step does not occur in any shortest path from s to t of the same length. Observe that during each augmentation step, any edge that had the minimum edge weight among the edges in the augmenting path will be saturated. This implies that there can be at most m augmenting paths used of length 1, and at most m augmenting paths used of length 2, and so on. If there is an augmenting path from s to t , there exists a simple augmenting path. Thus, the length of the longest augmenting path is at most $n - 1$. So after at most $m(n - 1)$ augmentation steps, a maximum flow will be obtained. The cost to find an augmenting path is $O(m)$, so the overall time complexity is $O(nm^2)$.

57.9 Further Reading

Ahuja, Magnanti, and Orlin [4], Even [51], Tarjan [146] provide good coverage of graph algorithms. Dijkstra's single-source shortest path algorithm [46] was published in 1959. The Bellman-Ford algorithm is based on algorithms independently developed by Bellman [20] and Ford [58]. Thorup developed an $O(m \log \log n)$ single-source shortest path algorithm [150], and an $O(m + n)$ time algorithm for undirected graphs with integer weights [149]. Ahuja et al. [5] presents an $O(M + n\sqrt{\log W})$ time algorithm for graphs with non-negative edge-weights, where W is the maximum edge weight. For graphs with negative edge weights, Gabow and Tarjan [70] present an $O(\sqrt{nm} \log(nW))$ time algorithm, and Goldberg [74] presents an $O(\sqrt{nm} \log W)$ time algorithm.

The first known minimum-spanning tree algorithm is due to Otakar Borůvka [27]. Borůvka's algorithm maintains a set $T \subseteq E$ of the spanning tree being constructed. Let $C_1, \dots, C_k$ be the connected components of the graph $G_T = (V, T)$. At each step, the smallest weight edge in each C_i to some other component is added to T . It can be proven that at most $O(\log n)$ iterations are required until T is a minimum spanning tree. Each iteration takes $O(m)$ time, leading to an overall time complexity of $O(m \log n)$. The algorithm that is commonly known as Prim's algorithm was first published by V. Jarník [90] in 1930, and reinvented by Prim [126] in 1957. Kruskal's

minimum-spanning tree algorithm [98] was published in 1956. The fastest known deterministic minimum spanning tree algorithm, by Chazelle [39], has time complexity $O(m\alpha(m, n))$ where α is the inverse Ackermann function (defined on page 80). The asymptotically fast minimum-spanning tree algorithm is the randomized algorithm of Karger, Klein, and Tarjan [92] that has $O(m)$ expected time complexity.

The Floyd-Warshall all-pairs shortest path algorithm was given by Floyd [54] in 1962, who based it on a theorem of Warshall [155] about computing the transitive closure of boolean matrices. Johnson [91] gives an $O(n^2 \log n + nm)$ all-pairs shortest path algorithm that is more efficient than the Floyd-Warshall algorithm for sparse graphs. Johnson's algorithm uses the result of the Bellman-Ford single-source shortest path algorithm to reweight the edges (when there is no negative weight cycle) in a way that makes all edge weights non-negative while preserving the shortest path. Then Dijkstra's single-source shortest path algorithm can be run from each vertex. Karger, Koller, and Philips [93] and McGeoch [110] have given an $O(nm^* + n^2 \log n)$ all-pairs shortest path algorithm for graphs without negative weight edges where m^* edges are on some shortest path.

Ford and Fulkerson [58] presented the Ford-Fulkerson maximum flow method. Edmonds and Karp [50], and Dinic [47] proved that this technique yields a polynomial time algorithm. Dinic introduced the notion of a **blocking** flow in which every path from the source to the sink contains a saturated edge. Using breadth first search a blocking flow can be computed in $O(nm)$ time, and at most $n - 1$ blocking flows must be computed before a maximum flow is obtained. Thus Dinic's algorithm has $O(n^2m)$ overall time complexity. The push-label maximum flow technique, which is very different than Ford-Fulkerson technique, was presented by Goldberg [73] and Goldberg and Tarjan [75]. They give both an $O(n^3)$ and an $O(nm \log(n^2/m + 2))$ implementation. Ahuja and Orlin [6] and Ahuja, Orlin, and Tarjan [7] gave maximum flow algorithms that use a scaling approach.

57.10 Quick Method Reference

AbstractWeightedGraph Public Methods

- p. 926** *AbstractWeightedGraph*(GraphRepresentation<V,E> rep, boolean multigraph)
- p. 947** ShortestPathMatrix<V,E> **allPairsShortestPaths**()
- p. 941** InTree<V,E> **generalShortestPathFromSource**(V source)
- p. 929** void **greedyTreeBuilder**(InTree<V, E> tree, double seedCost, Comparator<Double> comp)
- p. 939** Set<E> **kruskalMST**()
- p. 953** FlowGraph **maximumFlow**(V s, V t)
- p. 937** Set<E> **primMST**()
- p. 934** InTree<V,E> **weightedShortestPaths**(V s)

AbstractWeightedGraph Internal Methods

- p. 952** void **augmentFlow**(FlowGraph flow, V s, V t,
InTree<V, SimpleWeightedEdge<V>> residualFlowTree)
- p. 952** InTree<V, SimpleWeightedEdge<V>> **findAugmentingPath**(V s, V t, FlowGraph flow)
- p. 951** double **weightInResidualGraph**(V src, V dest, FlowGraph flow)

Part IV

APPENDICES

Appendix A

Java Fundamentals

This book presents data structures and algorithms using the Java programming language. Java is a strongly-typed object-oriented language that provides a garbage-collected heap. Java source code is compiled into byte code for execution on any platform that provides an implementation of the Java virtual machine (JVM). Executing programs within a virtual machine introduces some overhead, but provides the advantage of portability. The JVM also provides a flexible security management system that can be used to restrict the behavior of applications, such as those that might be downloaded from an untrusted source and executed within the JVM of a web browser.

The body of this book is written with the assumption that the reader has prior familiarity with fundamental concepts of the Java language and execution model necessary for a clear understanding of the software presented in this book. These concepts are summarized in this appendix and used heavily throughout the book. Other aspects of Java, such as portability and security, are omitted here since they are outside the scope of this book. Readers interested in additional details on the topics presented here, as well as other aspects of the Java programming language, should refer to *The Java Language Specification* [77].

This appendix assumes that the reader has programming experience in a procedural language (such as C), but it does not assume prior exposure to Java or object-oriented programming. Readers with significant object-oriented design experience in Java may wish to skim the appendix, but are strongly encouraged to read Chapter 1 carefully. Readers with significant object-oriented design experience in another language (such as C++) will benefit from reading this chapter in order to understand the relevant differences between the languages. Readers who already know Java may wish to consult this appendix to refresh their understanding of particular concepts or to clarify programming language terminology.

This appendix is organized as follows. We begin in Section A.1 by discussing the distinction between primitive types and user-defined types in Java. User-defined types are defined as classes, which are organized into packages, as discussed in Section A.2. Section A.3 describes how a class defines a type of object, which may encapsulate data and provide methods (Section A.3) to operate on that data. Objects are created through constructors (Section A.3) and are stored in an area of memory known as the heap (Section A.4). In Section A.5, we describe Java's exception handling mechanism, which affects the execution of both methods and constructors.

Sections A.7 and A.8 describe some necessary technical issues related to both nesting classes inside other classes and defining static members of classes. Section A.10 explains how to extend Java classes to create subclasses, and how inheritance and dynamic method dispatching support polymorphism. Finally, Section A.11 describes Java generics, a recent addition to the Java language that supports type checking for polymorphic data structures.

A.1 Types and Operators

Every value in Java has a **type** that defines its representation (how the data is laid out in memory) and its semantics (how values of that type are accessed and manipulated). For example, the *int* data type represents a sequence of 32 bits in memory that are interpreted as an integer using a

2's complement representation. **Operators** associated with the *int* data type, such as addition and multiplication, provide ways to manipulate values of that type. As in all programming languages, Java programmers write **expressions** to describe how values are to be combined. For example, if *i* and *j* are integers, then the value of the expression *i+j* is the result of applying the addition operator to the values of *i* and *j*.

Every type in Java is either a *primitive type* or a *user-defined type*. The **primitive types** are those that are built into the language. They are directly recognized by the compiler, and their operators are often supported directly in hardware. Some examples of primitive types include *int* (32-bit signed integers), *double* (64-bit floating point numbers), and *boolean* (which represent *true* or *false* values). The names of primitive types are keywords in the language, and the compiler recognizes constant values (such as -42, 3.14, and *true*) of these types. Primitive data types have associated operators that have special symbols within in the language, such as + (addition), <= (less than or equal to), and && (logical conjunction).

Java is strongly typed. Every variable is declared to be of a particular type and every operator (including programmer-defined procedures) is declared to have a particular (return) type. From this information, the compiler can conservatively deduce the type of every expression in the language. For example, when *i* and *j* are of type *int*, the compiler can determine based on the types of the variables and the operators, that the expression *i+j* is of type *int*, whereas the expression *i < j* is of type *boolean*. The compiler performs type-checking on every assignment to a variable to guarantee that the declared type of the variable is compatible with the type of the expression being assigned to that variable. In this way, programmers can rely on the fact that a variable will always hold a value of its declared type. A procedure that does not return a result is declared to have a *void* return type.

As do most programming languages, Java provides very few primitive types and operators. Richness in expressive power is obtained by combining these primitive types into **user-defined types**. Programmers define each new type as a **class**. Each value (or **instance**) of that type is called an **object**. Variables that refer to these objects are called **reference variables**. When a reference variable does not refer to any object, its value is **null**.

One may **extend** an existing class to create a new subclass in the **class hierarchy**. For example, the class *Box* could be extended to create the subclass *Cube* as a more specific type. In this case, *Box* would be considered the **superclass** (or **parent class**) of *Cube*. As will be discussed in Section A.10, subclasses **inherit** the implementation of the parent class, which enables significant code reuse. For the purposes of type checking, each instance of a subclass is also considered an instance of all of its ancestor types (the superclass, its superclass, and so on). The relationship is asymmetric: all cubes are boxes, but not all boxes are cubes.

It is important to note that the contents of a variable depend upon whether its type is a primitive or a class. When a variable's type is a primitive, then the contents of the memory location for that variable is the actual representation of a primitive value of that type. However, when a variable's type is a class, then the contents of the variable is not an object of that type. Instead, the variable holds an object **reference**, the memory address of the object. To use the object, the runtime system follows the reference to get the object at that memory location, a process known as **dereferencing**. **Sharing** occurs when two variables refer to the same object, in which case changes to the object made through one of the variables are visible through the other variable. Just as it does for variables, the value/reference distinction applies to the slots of an array. If the type of the array is a primitive, then its slots hold primitive values, but if the type of the array is a class, then its slots hold references.

When an object is created, Java allocates space for it in a section of virtual memory called the **heap**. The amount of space allocated depends upon the object's internal representation, as defined by the class. In some object-oriented languages, like C++, programmers must specify when an object should be destroyed. However, Java programmers do not specify when an object is no longer needed. Instead, Java's *garbage collector* automatically reclaims the storage occupied by objects that are no longer accessible from the executing program. This is discussed further in Section A.4, and in the case study of Section 53.9.

A.2 Packages

To support programmers in developing applications, the standard Java libraries include hundreds of classes, organized into **packages**. Such packages support constructing graphical user interfaces, managing files and network connections, etc. Besides organizing classes, packages partition the name space to prevent clashes. Two different packages may contain classes with the same name, and both classes may be used within the same application. This works because each class actually has a *fully qualified name* that includes its package name as a prefix for uniqueness. To be found by the compiler, the class files for each package must be stored in a directory with the same name as the package. The Java **class path** informs the compiler in which directories it can find the needed classes and interfaces. Package directories may be nested, in which case the package name mirrors the directory structure. For example, if directory *foo* contains directory *bar*, which in turn contains class *Baz*, then the package names are *foo* and *foo.bar*, and *foo.bar.Baz* is the fully qualified name of the class. To use *Baz* in an application, the classpath for both the Java compiler and the JVM must include the directory that contains *foo*.

A Java source file may **import** classes from other packages, as well as entire packages, for use within that file*. This allows the short class name, instead of the fully qualified name, to be used throughout the file. (In the rare case when a class name is defined in two imported packages, the fully qualified name is required to disambiguate each use.) All Java programs automatically import the *java.lang* package. Among other things, this package includes the *Math* class, which provides mathematical and trigonometric functions, and the *String* class, whose objects represent strings of characters that can be printed to the console using *System.out.println(String s)*.

Of particular relevance to Part II of this book, the package *java.util* contains classes for a variety of data structures, as well as some additional utility classes like *Date*, whose objects represent a calendar date and time. A good source for documentation on Java's standard library is java.sun.com/reference/api.

A.3 Classes and Objects

Building upon the primitive types and the Java libraries, application programmers define new types that support the information requirements and behaviors of their applications. The user-defined types are called **classes**. For example, an application that works with information about people might define a class called *Person*. The header of the definition would identify the name of the class, and the class definition would appear between braces following the header.

```
package mypackage;                                //the first line of the file names the containing package
import java.util.Date;

class Person {

}
```

In general, a class defines: **instance variables** defining the internal representation of the objects of that type, **methods** providing ways in which objects of that type can be accessed, manipulated, and

*While the *import* statements are not shown in the printed book, the provided CD does include them.

combined, and **constructors** defining how objects are initialized. In the following three subsections, we describe each of these through an illustrative example and a discussion of the syntax and related semantics.

A.3.1 Instance Variables

The **internal representation** of an object is the way in which information is stored within that object. For example, the class *Person* might include instance variables that hold a name, home address, and birthday as follows. Each of these variables is considered a **field** of the class. Observe that within one class definition, it is common to have variables of a user-defined type. For the purposes of this example, we assume the existence of a class *Place* whose objects represent street addresses.

```
String name;  
Place home;  
Date birthday;
```

A.3.2 Methods

The methods of the class *Person* might include simple **accessor** methods for retrieving particular pieces of information stored within the object's fields (such as the person's name) and **mutator** methods for assigning new values to fields (such as to set the person's home address). In addition, methods might be defined to perform computations.

We consider a simple example, and then describe the syntax and conventions used within this example. If the application requires sorting people by age, we would want to define a method in the *Person* class that determines if one person is older than some other person. When a method is called on an object, that method can use any of the instance variables within that object. In addition, the keyword **this** can be used as a reference to the object itself. For example, in the *setHomeAddress* method below, the keyword *this* is used to access the instance variable *home*, since its name is masked by the parameter of the same name. Note that the method delegates the actual computation to the *before* method of the *Date* class.

```
String getName() {  
    return name;  
}  
  
void setHomeAddress(Place home) {  
    this.home = home;  
}  
  
boolean isOlderThan(Person otherPerson) {  
    return birthday.before(otherPerson.birthday);  
}
```

By convention, we use lowercase letters for package names. We begin class names with uppercase letters, and we begin method names and variable names with lower case letters, capitalizing subsequent words that are used as part of the name. We also adopt the convention of prefixing each

accessor with the word “get” (or the word “is” if the return type is boolean), and prefixing each mutator with the word “set.” In the text of this book, we use a class name whenever referring to that class, but will sometimes use a lower case version of that name, with spaces between words, to refer to instances of that class. For example, when the context is clear, instead of saying “an object of type Person” or “instance of the Person class,” we might simply say “a person.”

Some classes do not provide any mutator methods. Instances of such classes are called **immutable**. Once created, the data stored inside of them never changes. A string is one example of an immutable object. Therefore, we often use a (mutable) *StringBuilder* object when we want to build up a string piece by piece. Other examples of immutable objects include the so-called **wrapper** classes that correspond to each of the primitive types. These wrapper classes include Integer (for *int*), Double (for *double*), Boolean (for *boolean*), etc. The purpose of these wrapper classes is to allow primitive values to be used where objects are expected, particularly in data structures that store objects. Since Java 5 automatically converts between instances of these wrapper classes and their corresponding primitive values depending upon the expected type of the expression.

The methods of the person example provide an illustration of the Java syntax for method declarations. Each method has a **header** that contains the return type (which may be *void*), followed by the method name and, in parentheses, a comma-separated list of **formal parameter** declarations. The method **body** is given between curly braces. Method bodies contain statements that are executed sequentially. Execution within the method body is controlled by a number of standard language constructs for conditionals (*if*, *else if*, and *else*) and loops (*for* and *while*). These have a standard syntax and semantics, similar to C and C++, so we do not elaborate on them here. Within statements, expressions are executed left to right, respecting the usual order of operations. In assignment statements, the expression to the right of the equals sign is executed to produce a value that is assigned into the variable identified on the left of the equal sign. Unlike some languages such as Pascal, “=” is used for assignment and “==” returns a boolean which is *true* iff the two variables are equal[†].

A **method call** is written using dot notation, as illustrated within the body of the *isOlderThan* method of the Person class above. The expression before the dot is a reference to the object on which the method is to be called. This object is known as the **target** of the method call. Following the dot is the name of the method and a list of the actual parameter expressions, whose values are to be passed into the method when it is invoked.

Control passes out from a method in one of two ways, by **returning** (discussed here) or by **throwing** (discussed in Section A.5). If a method’s return type is not void, it *returns* by executing a *return* statement with an expression that is evaluated to produce the *return value* of the method. If the return type of a method is *void*, it may return by executing a *return* statement (without a return value expression), or, if no return statement executes, the method terminates after the last statement in the body is executed.

Each execution of a method is called a **method invocation** and occurs as the result of a method call. After the target of the method call is dereferenced, each of the actual parameter expressions is evaluated. These values are bound to the formal parameters of the method, and the method body is executed. When a method returns, the return value (if any) is made available to the caller as the value of the method call expression. In other words, a method call expression evaluates to its return value.

The semantics of method invocation is **call by value** in which a copy of the value of the actual parameter is passed to the method. However, all variables of a user-defined types are references to objects. So a copy of the reference to the object is passed, not a copy of the object. So changes made to the object by the method remain after the method returns.

[†]Two references are equal only when they reference the same object. See Section 5.2.2 for further discussion on this topic.

The parameters, local variables, and return values of each method invocation are stored in an area of memory known as the **execution stack**, or simply *the stack*. More specifically, when a method is called, the stack grows to create an **activation record** (or *stack frame*) with enough space for all of the parameters and local variables that will be needed during execution of the method body. The parameter values are copied into the activation record, which serves as an environment in which the method will execute. When the method terminates, the activation record is discarded and the stack shrinks back down, leaving only the return value (if any) to be claimed by the caller. Note that since each invocation of a method results in a new activation record, there may be multiple activation records on the stack for the same method, each with its own values for the parameters and local variables. This arrangement enables recursion, as illustrated by Figure A.1, which shows the stack as it grows and shrinks during execution of a recursive method.

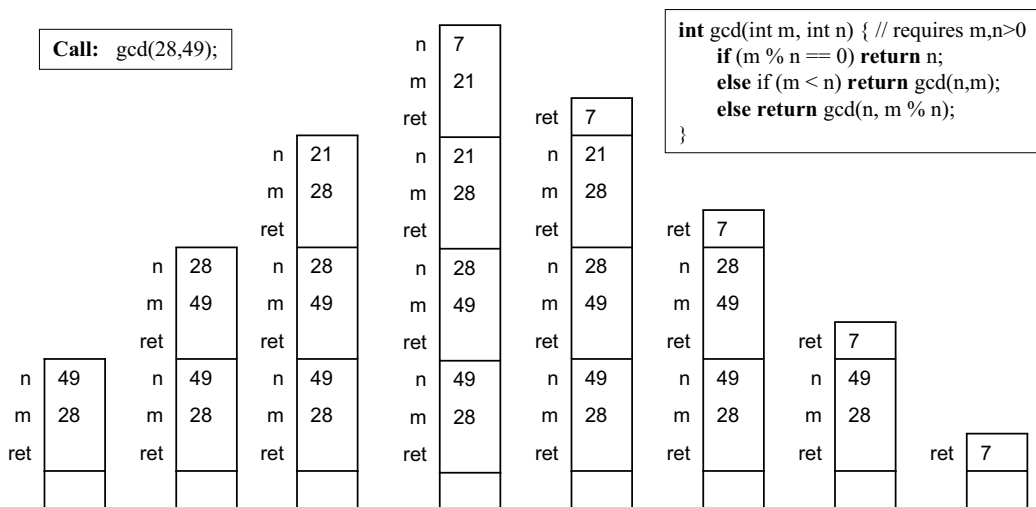


Figure A.1

The execution stack over time for a recursive implementation of Euclid's algorithm for computing the greatest common divisor of two positive integers.

Method names may be *overloaded*, meaning that more than one method of a class has the same name. In this case, the methods are distinguished on the basis of the parameter types. The return type is not used to distinguish methods, so one cannot define two methods in a class with the same name and parameter types, even if the return types differ. When the compiler encounters a method call using an overloaded method name, it attempts to disambiguate the call on the basis of the types of the actual parameters. If there is more than one match[‡], the most specific method is used. If there is not a most specific method, then the call is ambiguous and results in a compile error. (To determine which method is most specific, the compiler chooses the methods whose parameter types are further down in the class hierarchy.)

As a rule, the Java defines operator symbols (+, <=, &&, etc.) only for the primitive types, whereas all operations performed on objects occur via method calls as just described. One notable exception, however, is the String class, which is given special status by the Java compiler. In particular, the plus symbol (+), can be used as an infix operator on String objects to produce a

[‡]Multiple matches may occur when the type of the actual parameter is a subtype of the formal parameter type of two different methods. For a discussion of type checking in the context of class hierarchies, see [Section A.10.2](#).

new String object that is the concatenation of its two arguments. If only one of the two arguments to + is a String, then a String is produced from the other and then the two strings are concatenated. The String class also provides methods, such as *substring*, that can be called in the usual way.

A.3.3 Constructors

Objects are created by means of a **constructor**, which is a special method that is used to initialize the fields of an object. A constructor executes only once for each object, just after Java has allocated space for that object on the heap. There may be several constructors with different parameter lists, but all constructors have the same name as the class. Parameter values are typically used by a constructor to initialize fields of the newly created object. For example, consider the following constructors for the Person class.

```
Person(String name, Place home, Date dateOfBirth) {  
    this.name = name;  
    this.home = home;  
    birthday = dateOfBirth;  
}
```

```
Person(String name, Date dateOfBirth) {  
    this(name, null, dateOfBirth);  
}
```

The second constructor, which takes only two parameters, calls the other constructor using the keyword *this*, and uses *null* as the default value for home. It is good design practice to have only one constructor that does the “real” initialization work, because it avoids duplication of code and simplifies the other constructors that may be provided for convenience of the users of the class.

Objects are created by calling a constructor using the *new* operator. When a constructor is called, the value of the “new” expression is a reference to the object created. For example, executing the statement

```
Person p = new Person(“Sam”, new Date(1903, 12, 1));
```

would create a person named “Sam” with a birthday of December 1, 1903. Note that two objects are created by this statement, the Person object (whose reference is retained in variable *p*) and the Date object (whose reference is passed to the constructor and saved in the *birthday* field of the Person object).

A.4 The Heap

As mentioned earlier, when an object is created, Java allocates sufficient space for it in a section of virtual memory called the **heap**. Each reference to that object actually contains the address of the object in the heap. For example, consider the following method.

```
Person testPerson() {  
    Place a1 = new Place(“4 Privet Drive”, “Surrey”, “England”);
```

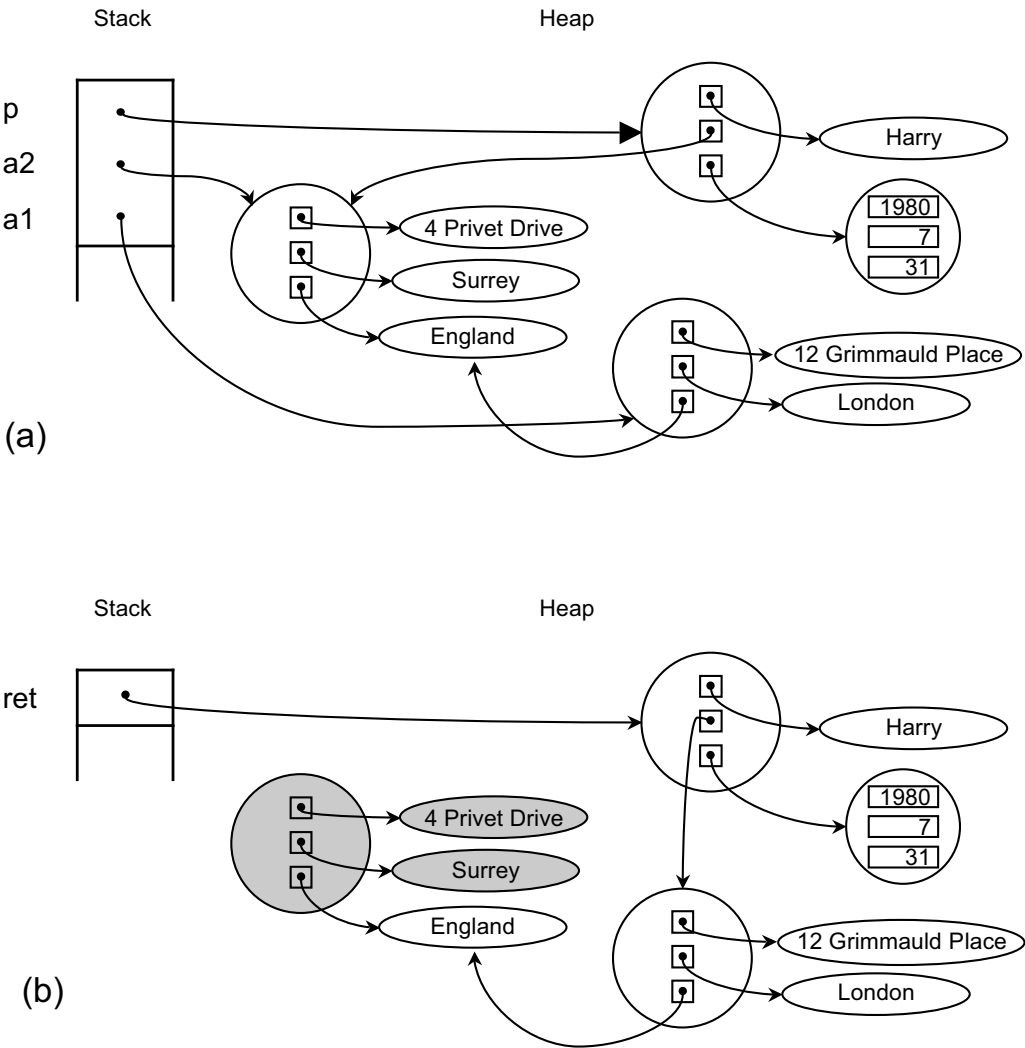



Figure A.2
The stack and heap during and after execution of the `testPerson` method. The garbage is shown in gray.

```

Place a2 = new Place("12 Grimmauld Place", "London", "England");
Person p = new Person("Harry", a1, new Date(1980, 7, 31));
p.setHomeAddress(a2);
return p;
}

```

Part (a) of [Figure A.2](#) shows the stack and heap immediately after execution of the third line of the `testPerson` method. Notice that the variable *b* on the stack refers to the `Person` object, which in turn refers to a `Place` object.

We have said that Java programmers do not directly specify when an object is no longer needed. The Java designers ruled out explicit object destruction as a dangerous violation of type safety, since it would admit the possibility of *dangling references* that appear to refer to an object of a particular type, but instead refer to a memory location containing arbitrary data.

However, it is important that space occupied by unneeded objects be reused for new objects, or a long-running application would eventually run out of memory. For this purpose, Java provides a garbage collection algorithm that runs periodically to remove unneeded objects from the heap. In general, it is impossible to tell whether an application might eventually dereference a reference that it is holding. Therefore, garbage collectors adopt a conservative approach that retains objects in memory as long as they are “reachable.” More specifically, an object is considered **reachable** by an executing program if there is a reference to that object on the execution stack, or if some other reachable object refers to that object. All other objects are considered to be **garbage**.

To see how an object might become garbage, consider part (b) of [Figure A.2](#), which shows the stack and heap immediately after execution of the `testPerson` method. The return value on the stack refers to the `Person` object, which in turn refers to the `Place` object for “12 Grimmauld Place,” but there is no longer any way to reach the `Place` object for “4 Privet Drive” by following any chain of references from the stack. Therefore, there is no way for the program to ever again refer to that object, so the objects shown in gray are considered garbage.

The garbage collector does not necessarily collect garbage immediately, because aggressively detecting garbage would add too much overhead to program execution. In fact, if a program uses very little memory, the garbage collector may never run.

From the perspective of this book, it is important to remember that while execution of methods occurs in the stack, all objects reside on the heap. Because reachable objects persist on the heap, it is important to remove from data structures any references to objects that are no longer needed, or else they will not be considered garbage and will continue to waste space in memory. Finally, creating many “throw away” objects results in more garbage collection overhead. One way to reduce this overhead is described in [Section 5.5](#), where we define a pool of reusable objects as a mechanism for avoiding the continual creation of fresh objects that may be subsequently garbage collected.

A.5 Exception Handling

An exception is an object whose purpose is to describe an unusual or unexpected situation that has occurred during execution. All exceptions are instances of the `Exception` class and its subclasses.

One important subclass of `Exception` is `RuntimeException`. A runtime exception generally indicates a mistake in the program. The subclasses of `RuntimeException` include, for example, `NullPointerException` (which indicates an attempt to dereference a *null* reference), `ArrayIndexOutOfBoundsException` (which indicates an attempt to access an array slot using an index that is negative or greater than the index of the last array slot), and `IllegalArgumentException` (which indicates that

the caller of the method has supplied a parameter value that does not satisfy the requirements of the method).

Any other exception (one that is not an instance of *RuntimeException*) describes a situation that may not necessarily indicate an error, but that cannot be handled by the part of the program that has discovered it. In the following illustration, the *open* method throws a *FileNotFoundException* if an attempt is made to open a file that does not exist. Note that the constructor of the *FileNotFoundException* class takes a *String* as a parameter, which is used to store additional information inside the exception object.

```
InputStream open(File f) throws FileNotFoundException {
    if (!f.exists())
        throw new FileNotFoundException(f.getName());
    else
        return new FileInputStream(f);
}
```

When a method terminates by throwing an exception, the exception object is placed on the stack instead of a return value. Control is passed back to the caller by means of the exception handling mechanism. The syntax for handling an exception is shown below. The **try block** surrounds statements that might throw an exception. When an exception is thrown due to any statement in the try block, the rest of the try block is not executed. Instead, control moves to the first corresponding **catch block** whose declared exception variable is assignable from the type of exception that was thrown.

```
try {
    InputStream is = open(new File("filename.txt"));
    //use the input stream
} catch (FileNotFoundException fnfe) {
    //take corrective action
}
```

The application code that handles the exception has been omitted from the example. The application might choose to prompt the user for a new file name, create a new file, or take some other action. The important point to recognize is that the *open* method has no way of knowing what the application would like to do when an error occurs. Therefore, it throws the exception to inform the caller, which then has the option to handle the exception in whatever manner it deems appropriate.

Although not illustrated here, multiple catch blocks may follow a single try block. In that case, the first one that matches the type of the exception thrown will be the one that executes. If no matching catch block is found, the exception propagates to the catch blocks of the surrounding try block, if any. If a method does not handle a thrown exception, the exception propagates to its caller, and so on, until the exception is either handled or the entire execution stack is depleted, in which case a stack trace showing the exception is typically printed to the console.

Exceptions of type *RuntimeException* (and its subclasses) are **unchecked exceptions**, while all other exceptions are **checked exceptions**. If a method has the possibility of throwing a checked exception, then it must declare that fact in its method header. For example, the *open* method above declares that it throws the *FileNotFoundException*. If a method declares an unchecked exception in its header, then every method call to that method must be within a *try* block that has a *catch* block that handles that exception, or else the method containing that method call must also declare that it throws the exception. The purpose of this restriction is to guarantee that the caller of a method is always aware that such exceptions may be thrown. No such restriction is made for unchecked

exceptions (*RuntimeExceptions*) because these generally should not be caught and handled; instead, they indicate errors in the program that should be fixed to prevent the exceptions from occurring.

A.6 Assertions

Throwing and catching exceptions, as described above, is the most common mechanism for preventing misuse of methods. For example, one might throw an *IllegalArgumentException* if the parameter requirements of a method are not satisfied. However, sometimes testing the conditions under which an exception may be thrown can be expensive, and in some cases it may be undesirable for the exception to be thrown except during software development and testing. Java's **assert** statement provides an alternative way to check conditions at run time, provided that the JVM is launched with its *assertionsenabled* flag turned on. For example, suppose that an application will issue a request to deliver a package by courier only if there is a short path from the warehouse to the customer in a street map represented as a graph. However, suppose that before calling the method, it should be known that this short path exists. One could check again inside the method, but that may unnecessarily cost. Instead, one could use an assert statement, as shown below, to check the condition during testing.

```
void deliverProductByCourier(Place warehouse, Place customer) {  
    assert graph.shortestPath(warehouse, customer) < MAX_COURIER_DISTANCE;  
}
```

Once the code is sufficiently tested, assertion checking could be turned off in the JVM without modification to the code, and the extra time to perform the checks would be avoided. Without careful thought, turning off error checking can be dangerous and is warranted only when the code will undergo rigorous testing before deployment and when the checks would cause significant overhead during execution. We use normal exception handling throughout most of the code in this book. However, there is one illustration of the use of an assertion in the *addEdge* method of *AbstractGraph* on page 860.

A.7 Inner Classes

When a particular class uses closely related types, it is sometimes convenient to define those types as **inner classes** within the class itself. Each instance of an inner class implicitly contains a reference to an instance of the surrounding class, so the inner class object can directly use the instance variables and call the methods of that object.

Java supports three kinds of inner classes: member classes, local classes, and anonymous classes. **Member classes** are defined directly within the curly braces of the class definition. Like instance variables and methods, access modifiers can be used to specify the visibility of a member classes. Member classes can be instantiated and used anywhere within the containing class and its other inner classes. Most pointer-based data structure have an inner class that holds the objects that comprise the data structure. For example, a *BinaryTree* class might use an inner class to represent its tree nodes. (See Section 1.6 for a discussion of tree representations.)

```

class BinaryTree {
    Node root;                                //reference to the root of the tree

    class Node {
        Node leftChild;                        //reference to left child
        Node rightChild;                       //reference to right child
        Node parent;                           //reference to parent
        Object data;                           //reference to associated data

        //constructor and methods for Node class
    }
}

```

Local classes are defined within methods. They can be instantiated only within those methods and have access to any parameters and local variables of the class that are declared *final*. Finally, **anonymous classes** are defined within an expression using a special syntax, and are instantiated as part of the evaluation of that expression. Examples of member classes can be seen throughout the book, particularly to define auxiliary classes that are used as part of the internal representation of data structures. Section 14.5 includes three anonymous classes. We do not use any local classes in this book.

A.8 Static Members

So far we have discussed four kinds of members: instance variables, methods, constructors, and member classes. We think of constructors as part of a class, whereas instance variables, methods, and member classes are accessed through objects. For example, we do not need an object to create a new instance of a class, but we need an object to access one of its instance variables or call one of its methods.

Sometimes we want to define members of the class itself. For example, we may want to define a *constant* that is the default initial capacity of a set. Since the value is a constant, we do not need a copy inside of every object. In fact, that would be a waste of space. Therefore, we could define the constant as follows.

```
public static final int DEFAULT_CAPACITY = 8;
```

There are several things to notice about this declaration. First, the *static* modifier means that there is only one, and it is part of the declaring class itself. In other words, this is *not* an instance variable. Second, declaring the variable as *final* means that the compiler will verify that no statement will change its value. Consequently, we are free to make the variable *public* without fear of external modification. (However, we may choose not to make it public if we do not wish users to know or depend upon its value.) Finally, notice that we follow the Java convention of using all upper case letters in constant identifiers, to distinguish them from ordinary variables.

When the **singleton design pattern** is used, often a static variable is created. (See Section C.14 for more discussion of this design pattern.) For example, in the `OpenAddressing` class (Chapter 22) the static variable `REMOVED` is used for all hash table slots in which an element had been removed.

Static variables need not be a constant. For example, if one want to assign a unique id to every instance instantiated for a class, then a static variable:

```
public static int idCount= 0;
```

would be defined. Then the constructor could include:

```
id = idCount++;
```

which both sets the id to the current count, and increments the count.

The *static* modifier can also be applied to methods. A **static method** is essentially a *stand-alone procedure* that does not have access to the instance variables and non-static methods defined in the class. Static methods are useful for defining simple functions that neither operate upon nor are associated with objects of the class. For example, the class `Math` in the package `java.lang` provides many static methods for mathematical and trigonometric functions. There is no need to have an instance of a class to call its static methods. Instead, one refers to the class itself using the familiar dot notation. For example, the call `Math.abs(x)` would return the absolute value of `x`.

Marking a member (inner) class as *static* changes the class in two ways. First, each instance of the **static class** will no longer have an implicit reference to an object of the containing class, so the inner class cannot reference instance variables and non-static methods. Second, the static class can be instantiated from outside, calling its constructor using a name that is prefixed by the surrounding class name and a dot. For example, if the `Node` class above were static, one could instantiate it from outside the class using the expression

```
new Tree.Node();
```

Note, however, that if an inner class is not static, it is a member class, which implies access to the instance variables of an object of the containing class. Therefore, a member class must either be instantiated within a method of an object of the containing class, or its constructor call must be preceded by a reference to an object of the containing class. For example, the `Locator` class is an inner class of the `Array` class. The `SortedArray` class (Chapter 30), which wraps the array class, has the statement

```
Array.Locator loc = a.new Locator(Array.Locator.FORE);
```

where `a` is an array. This statement allocates a new array locator for Array `a`, where the locator is initially at the position represented by the static constant `FORE` which is defined within the array locator inner class.

A.9 Interfaces

A Java **interface** is a list of method signatures without method bodies. An interface has no variables and no constructors. It merely defines a type by listing the methods that are available on objects of that type. For example, the following interface is for an ADT for maintaining a set of elements.

```
public interface Set {
    public void add(Object x);           //adds object x to the set
    public void remove(Object x);       //removes object x from the set
    public boolean contains(Object x);  //true when object x is in the set
    public boolean isEmpty();           //true when the set is empty
}
```

The interface *Set* defines a new type, just as the class *Person* defines a type. An application program can declare variables of type *Set* and call methods through those variables. For example, the following method takes a *Set* as a parameter and adds several elements to the set. Note that any object of type *Set* can be supplied as a parameter of the method, and if there were multiple different implementations of the set ADT, any of them could be used.

```
void addPeople(Set buddySet) {
    buddySet.add(person1);
    buddySet.add(person2);
}
```

Interfaces cannot be instantiated since they have no constructors or method bodies. To obtain an object of type *Set*, one must call a constructor on a class that **implements** that interface. A class that implements an interface declares that fact in its header, as shown below. During compilation, the compiler checks that the class provides implementations of the methods in the interface.

```
class MySet implements Set {

    public MySet() {
        //constructor
    }

    public void add(Object x) {
        //implementation of add
    }

    public void remove(Object x) {
        //implementation of remove
    }

    public boolean contains(Object x) {
        //implementation of contains
    }

    public boolean isEmpty() {
        //implementation of isEmpty
    }
}
```

Each instance of the class *MySet* is of type *Set*, as well as of type *MySet*. Therefore, one may call a constructor of *MySet* and assign the result to a variable of type *Set*.

```
Set s = new MySet();
```

Consequently, an application program using sets could be written entirely in terms of the set ADT up until the point at which an actual object must be created. At that time, a particular class implementing that ADT must be selected and instantiated.

A.10 The Class Hierarchy

At the beginning of this chapter, we said that one may *extend* an existing class to create a new subclass in the *class hierarchy*. Each class in Java has exactly one parent class, except for the class `Object`, which is at the top of the class hierarchy and is the parent for all classes that do not specifically extend some other class. By default, a child class inherits the behavior of its parent, but this behavior can be selectively extended and overridden to create a more specialized type. A class inherits all of the implementation defined in its parent and, transitively, its other ancestors. The subclass may use any of the variables and methods defined in the parent class, in accordance with any visibility restrictions imposed, as discussed in the access modifiers subsection (Section 1.2). Although a class is limited to one parent, a class may simultaneously implement any number of interfaces.

A.10.1 Extending Classes

When extending a class, the parent class appears in the class declaration. For example, the following class `Cube` extends `Box` and implements two interfaces **Serializable** and **Cloneable**.

```
class Cube extends Box implements Serializable, Cloneable {
    public Cube(int side) {
        super(side, side, side);
        //the rest of the constructor implementation
    }
}
```

A.10.2 Type Checking and Casting

We think of a subclass as having a more specialized type than its parent, but each instance of a subclass is also an instance of the parent type. Therefore, when the compiler performs type checking, it permits an expression of a subtype to be assigned to a variable of any supertype. A **cast** instructs the compiler to treat an expression of one type as another type. When assigning an expression of a supertype to a variable of a subtype, a cast is required because the compiler cannot verify that the result of the expression will, in fact, be the more specific subtype. When a cast is used, the compiler inserts code to perform a runtime check that will result in a *ClassCastException* if the object does not turn out to be of the correct type. Finally, when the compiler can determine that an expression cannot possibly produce a value of the right type, then it will generate an error even if the programmer attempts to cast that value. The following code provides an example of each case.

```
Box b = new Cube(3);           //acceptable, because every cube is a box
Cube c = b;                   //won't compile, c's declared type is more specific than b's
c = (Cube) b;                 //will compile and run, because b actually does refer to a cube
b = new Box(1, 2, 3);         //clearly acceptable, since the types match
c = (Cube) b;                 //compiles, but throws a ClassCastException since b refers to a Box
String s = "test";           //again acceptable, since the types match
c = (Cube) s;                 //won't compile, even with cast, since s can't refer to a Cube
```

Note that casting of object references concerns type checking only. That is, a cast of an object reference neither results in a different value nor changes the type of the object. This is different

from casting of primitives, where a new value is produced. For example, in the following statement, the variable *i* takes on the value 3 after the cast.

```
double d = 3.7;
int i = (int) d;
```

A.10.3 Superconstructors

If a class lacks a no-argument constructor, then the first statement in each constructor in every subclass is required to be a call to a constructor from the parent class, known as a *super constructor*. This ensures that the fields defined in the parent class are initialized before the rest of the constructor executes. This is accomplished using a special *super* syntax, which is distinct from simply saying *new Box(...)*, which would result in creating an additional object. In the above example, the *Cube* constructor calls the *Box* constructor. Note that the parameters to the child constructor may differ from those of the parent. Here, the *Cube* takes one parameter (the length of a side), which is passed to all three parameters of the *Box* constructor (presumably the length, width, and height).

A.10.4 Overriding Methods

In addition to those that are inherited, subclasses may define variables and methods to provide additional functionality within the subclass. In addition, a subclass may **override** a method of the parent class by repeating the method header and substituting a different method body. This is useful when the subclass needs to perform a more specialized computation than the one that is inherited.

Method dispatching occurs dynamically, on the basis of the type of the object, not the type of the variable or expression that refers to the object. For example, if a variable *b* of type *Box* actually refers to an object of type *Cube*, then when a method is called on *b*, the method defined in (or inherited by) the *Cube* class will be executed.

Sometimes, it is desirable to override a method in order to perform some computation in addition to that carried out by the inherited method. Similarly, one may only want to use the inherited method with a slight change in the parameter values. To avoid repeating the body of the inherited method, one can call it from within the overriding method using the keyword *super* as in the following example.

```
int myMethod(int x) {                                //overrides an inherited method
    computeOffset();                                //do some preliminary work to set a variable, offset
    super.myMethod(x + offset);                      //call inherited method with different parameter
}
```

A.10.5 Polymorphism

Dynamic method dispatching gives rise to *polymorphism*, a powerful feature of object-oriented programming. With polymorphism, a program written in terms of general types can be extended by introducing new subtypes that override certain methods. The original code, which was written without any knowledge of these new subtypes, makes method calls through variables of the general types. However, when those variables contain instances of the new subtypes, their (overriding) methods are run, making the program behave differently in accordance with the new types. For example, a program might be written in terms of objects of type *BankAccount*. Later, a class called *RetirementAccount* might be created that overrides the *deposit* and *withdraw* methods inherited from *BankAccount* in order to manage certain legal restrictions on retirement accounts. However,

code in the original program can continue to work as before. Whenever one of its `BankAccount` variables happens to refer to a `RetirementAccount`, polymorphic method dispatching will guarantee that whenever the original code performs deposits or withdrawals, they will exhibit the correct behavior in accordance with their types, without any special treatment in or modification of the original code. Polymorphism allows each specialized subtype to “take care of itself.”

Since polymorphism depends upon all methods of the general type being available on all subtypes, a subclass may not “take away” a method. Provided that the method is not final, a subclass may override the method to do nothing or throw an exception, but the subclass must provide some implementation of the method. Consequently, any overriding method must have a visibility level that is at least as permissive as that of the method it overrides. For example, a class cannot override a public method with a private one.

Use of interfaces, as described in Section 1.4 is another form of polymorphism. Each object whose class implements a particular interface may be used anywhere a variable of that interface type is required, and method dispatching will guarantee that the appropriate method (as defined in that class) will be executed.

A.10.6 Iterators

One of the advantages of object-oriented programming is encapsulation. For example, consider an implementation of a list data structure involving a `ListItem` class, where each `ListItem` object contains a reference to one of the user’s data objects and a reference to the next `ListItem` object in the data structure.

In such an implementation, it is desirable to keep the internal representation hidden from the user. Exposing the representation could result in unwanted external modifications to the references within the list structure, violating the representation invariants. Furthermore, exposing the representation would limit the data structure designer’s flexibility in changing that representation later, since applications may be written to depend upon details of the representation that was originally exposed.

Now consider the problem of traversing the list. Using an unencapsulated data structure, one might define a variable that refers to the first list item, and then loop through the items, advancing the reference variable to the next item after each iteration of the loop. Such a solution is efficient, since traversal is accomplished in time proportional to the length of the list. However, this implementation exposes the internal representation. In particular, the user needs a reference to the data structure’s internal `ListItem` objects.

To keep the internal representation hidden, one could imagine a solution in which the list provided a method `get(i)` that returned the i^{th} entry of the list. With such a method, the user could define a loop that traverses the list by calling `get` repeatedly with increasing values of i . However, such a solution is likely to be inefficient, since each call to `get` would presumably need to traverse the list up to the i^{th} element. Thus, the overall traversal would require quadratic time, whereas linear time performance is desired.

An **iterator** is an object that encapsulates a reference to an object in the internal representation of a data structure. The iterator is provided to the user, who can invoke methods to traverse the data structure, see the user elements in the data structure, and perhaps remove some of them. The iterator keeps moving the reference forward as the user requests, but it does not expose the reference itself to the user. Java defines an *Iterator* interface with the following public methods.

boolean hasNext(): returns true if there are more elements (i.e., the traversal has not reached the end)

Object next(): returns the next data element, advancing the iterator

void remove(): removes the data element that was just returned

With this interface, and an *iterator* method in the list class that returns an iterator object, the user could traverse a list containing String objects as follows.

```
Iterator it = myListOfStrings.iterator();
while (it.hasNext()) {
    String s = (String) it.next();
    //process s
}
```

A common mistake in the use of iterators is to call *next* multiple times within the same iteration of the loop. Note that each call to *next* advances the iterator, so it is necessary to save the result of the call in a variable if the return value is to be used in more than one expression.

The following alternative syntax is available since Java 5 for iterating over objects whose classes implement the *Iterable* interface, meaning that they provide an *iterator* method.

```
for (Object x : myListOfStrings) {
    String s = (String) x;
    //process string s
}
```

Since the return type of *next* is *Object*, it is often necessary to cast the result to the actual type of the objects returned. Java 5 solves this casting problem with parameterized types (**Java generics**), as discussed in Section A.11.

A.11 Generics

Most general-purpose collections of objects are *polymorphic*, meaning that they allow objects of different types to be included in the collection. For example, the set of objects discussed in Section 1.4 allows any object to be inserted into the set. However, applications that use polymorphic collections often expect values to be of a particular type. For example, an application may require a set of Integer objects. A general set of Objects would suffice for this purpose, but it would provide no type checking on the values inserted into the set. Furthermore, each time an element is returned from a collection of objects, or from an iterator over the collection, the return type of the method is *Object*, so casting is required to treat it as an *Integer*. This is acceptable as long as the programmer is very careful, but the lack of type-checking means that a programming error could result in a non-integer object being inserted into the collection. Such errors are problematic for two reasons. First, instead of being caught at compile time, they occur at run time (as a *ClassCastException* when an attempt is made to treat the non-integer value as an *Integer*) and can only be discovered through testing or program verification. Second, the cause of such errors can be difficult to find because the erroneous line of code (where the value of the wrong type is improperly inserted) is often far away from the line of code at which the problem surfaces (as a *ClassCastException* when an attempt is made to use the improperly inserted value).

Therefore, when using a polymorphic collection type, it is often desirable for a particular collection instance to restrict the types of objects that may be inserted. One could imagine defining a separate class (e.g., *SetOfIntegers*) for each required data type, but this would result in a proliferation of classes that essentially serve the same purpose, but for different types of elements. A better

solution, which is provided in Java 5 and Java 6, is to allow a polymorphic data type to accept one or more **type parameters**, and let each declaration and instantiation supply the types to be used. So, one can define a class `Set<E>`, and then declare and construct instances of that class that specify the particular type for `E`, such as `Set<Integer>` or `Set<String>`. Since the class `Set<E>` can use `E` as a type throughout the class definition, methods may accept parameters of type `E` or produce return values of type `E`. In Java, such parameterized data types are known as *generic* types. In the `Set` interface below, the existence of a type parameter allows the compiler to check that only elements of the proper type are inserted into the set, and also eliminates the need for casting when elements are returned from the set. Notice that the `iterator` method returns an `Iterator<E>`, whose `next()` method would have `E` as the return type, avoiding the need to cast the return value.

```
public interface Set<E> implements Iterable<E> {
    public void add(E x);                                //adds object x to the set
    public void remove(E x);                             //removes object x from the set
    public boolean contains(E x);                         //true if object x is in the set
    public boolean isEmpty();                             //true if the set is empty
    public Iterator<E> iterator();
}
```

If `MySet<E>` implements `Set<E>`, then one could write the following lines of code to create a set of integers, insert nine values into the set, and then iterate over the elements of the set to find the sum of the elements.

```
Set<Integer> s = new MySet<Integer>();
for (int i = 1; i < 10; i++)
    s.add(i);
int sum = 0;
for (int i: s)
    sum += i;
```

When a generic class or interface is defined, it may provide restrictions on its type parameters. For example, a collection may require that all inserted elements to extend a particular class or interface `C`, presumably because the implementation of the collection calls methods of `C` on its elements. To accomplish this, one may restrict the element type of the collection, as follows. Throughout the code, methods of `C` can be called on elements of type `E` without the need to cast.

```
public class MySet<E extends C> {
    public void add(E x) { ... }
}
```

One may also parameterize the types of the arguments to methods. For example, a constructor for `MySet<E>` may allow a `Comparator` to be provided for the purpose of establishing an ordering on the elements of the collections. It is necessary that the `Comparator` be capable of sorting elements of type `E`, but a more general comparator (such as one that could sort objects of some supertype of `E`) would also suffice. Therefore, one could allow such generality by specifying that a `Comparator` over any type that is `E` or a supertype of `E` is acceptable:

```
public class MySet<E> {
    public MySet(Comparator<? super E>) { ... }
}
```

Appendix B

Complexity Analysis

A primary goal of this handbook is to guide a reader with a specific problem to the most appropriate data structure(s) and the most appropriate algorithm(s) for a particular application. Generally, the most important criteria in this selection process is the efficiency, in time and space, of the solution. Therefore, a natural question arises: How does one measure the efficiency of an algorithm or data structure method? As a simple example, we consider the problem of sorting a collection of n comparable elements. The algorithm's execution time depends on many things, including:

- The hardware on which the algorithm will be executed.
- The compiler used create the executable.
- The size of the input (i.e., $n = 1000$ versus $n = 1,000,000$).
- The form of the input itself. For example, if insertion sort is given an input that is nearly sorted, it runs very quickly, yet it runs much more slowly on an input in reverse sorted order.

Suppose that several different algorithms to solve a desired problem have been proposed. How can they be evaluated to select the fastest one? One solution is to implement the proposed algorithms and compare the execution time that each takes on a selected set of inputs. This leaves the following decisions:

- What programming language should be used for the testing?
- What compiler should be used?
- What hardware and operating system should be used?
- What data should be used for the testing, and how should this data be selected?

We use the term **platform** to refer to system specific choices such as the hardware, operating system, programming language, and compiler. Clearly differences in the platform can have a significant affect on the execution time for an algorithm. However, if algorithm A is twice as fast as algorithm B when run on input x on one platform, generally algorithm A is roughly twice as fast as algorithm B for the same input on a different platform, unless there is hardware support that one algorithm uses and the other does not.

While sometimes the particular platform must be considered in the process of selecting the most efficient algorithm or data structure for a problem, generally it need not be considered. So a methodology is needed to evaluate the efficiency of a variety of algorithms independent of the platform. Such a methodology, known as **machine independent**, could be helpful but how can a machine independent measure of time complexity be defined? What can be measured?

A good machine independent quantity of computation time is the number of statements of code executed under a model of computation that is fairly consistent with the hardware. One could use the Java virtual machine for this purpose. While the execution time for one line of code might be very different than the execution time for another line of code, in practice the ratio between the

number of lines of code executed between algorithm A and algorithm B on the same input correlates very well with the ratio between the execution time of algorithm A and algorithm B.

Regardless of the model of computation, exactly counting the number of statements of code executed would be very tedious. So instead a rough measure that can be used to evaluate when one algorithm clearly dominates the other, particularly as the input size increases, is preferred. Section B.2 presents **asymptotic notation**, which provides a way to categorize mathematical functions based on rate of growth of the functions as n approaches infinity. The advantage of such a rough measure is that it can be calculated even for complex algorithms, and provides a way to determine which algorithms will scale best without the need to implement the algorithm, and in a way that takes into account all possible inputs.

B.1 Time Complexity

As a way to measure the dependence of the time complexity on the input size, we define the **time complexity** function $T_A(x)$ to be the number of statements executed by algorithm A on input x . We use the $|x|$ to denote the size of x as measured by the number of bits to encode x . Informally, $|x|$ measures the size of the input. For example, if A is an algorithm to sort n comparable elements, then $|x| = n$.

The time complexity of an algorithm can vary significantly between inputs x_1 and x_2 even if $|x_1| = |x_2|$. Thus, some method is needed to account for this dependence on the particular input. The most common solution is to define $T(n)$ to be an upper bound on the number of statements executed for any input x where $|x| = n$. More formally, we say that algorithm A has **worst-case time complexity** $T_A(n)$ when for each possible input x with $|x| = n$, $T_A(x) \leq T_A(n)$.

For some problems, the time complexity can be quite high for a small number of relatively unlikely inputs, but for the inputs that typically occur, the time complexity is much less. In such cases, the average time complexity is often a preferable measure. To formally define any notion of average time complexity requires specifying a probability distribution over the inputs. In such settings, the expected time complexity (where the expectation is taken over the random selection of the input) is a good measure. A **randomized algorithm** is an algorithm in which the execution for a particular input varies based on the outcome of a random number generator. For a randomized algorithm A , we say that A has **expected time complexity** of $T_A(n)$ when for each possible input x for which $|x| = n$, $E[T_A(x)] \leq T_A(n)$. Section B.4 discusses the expected time complexity in more depth. Example uses in this book of randomized analysis include the quicksort algorithm (Section 11.4.5), the selection algorithm (Section 11.5), and the the skip list data structure (Chapter 38).

Another common measure is **amortized complexity**, which is used when an operation is usually very efficient, but on occasion the operation reorganizes the data structure, causing it to run much slower. For example, it takes constant time to insert a new element at the end of a dynamic array (Chapter 13), as long as there is an empty slot in the array. However, when the array is at capacity, a new larger array must be allocated and all of the elements must be copied into the new array, before the new element can be added. When this resizing occurs, the cost of adding an element is linear, even though it is usually constant. However, the total worst-case time complexity for adding n elements (including the resizing) grows proportionally to n . Thus, the average cost per operation is a constant. The amortized cost is defined as the average cost per operation for the total worst-case cost of the entire sequence of operations. Section B.5 defines the amortized complexity more formally, and discusses techniques to compute the amortized complexity.

Finally, Section B.6 presents a simple fairly general technique to compute the asymptotic time complexity for recursive algorithms designed using a strategy called divide-and-conquer. The divide-and-conquer strategy is discussed in Section 11.4.2 during the presentation of merge sort.

B.2 Asymptotic Notation

The ability for an algorithm to scale well as the input size grows large is one of the most goals in the design of good data structures and algorithms. However, exactly computing the worst-case number of statements $T_A(n)$ executed by A on an input of size n would very tedious. **Asymptotic analysis** provides a way to to separate all algorithms into groups based on their performance, and **asymptotic notation** provides a language to discuss the behavior of mathematical functions as n approaches infinity.

As a motivating example, suppose that algorithm A_1 can solve any problem of size n by executing at most $100n \log_2 n$ statements, and that algorithm A_2 can solve any problem of size n by executing at most $5n^2$ statements. Is this enough information to say which algorithm is better? Figure B.1 shows $6n \log_2 n$ and n^2 as a function of n . When $n \leq 29$, $n^2 < 6n \log_2 n$. However, when $n \geq 30$, $6n \log_2 n < n^2$. Furthermore, the difference between these two functions grows further apart as n increases. So clearly algorithm A_1 is the better choice based on the efficiency. If the constant of 6 is made larger, the same behavior occurs, the crossing is just at a larger value of n .

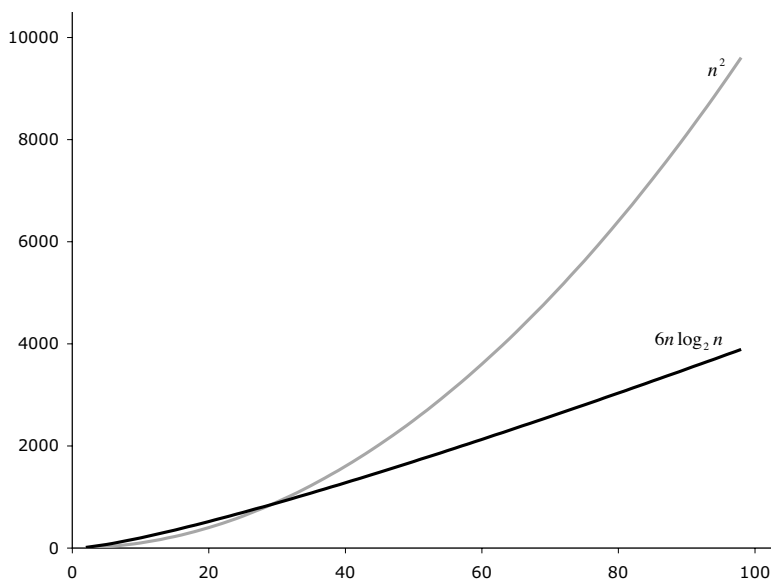


Figure B.1

A plot comparing the functions $6n \log_2 n$ and n^2 as n increases.

For small input sizes (e.g., $n < 100$), most solutions will run very quickly, so the algorithm and data structure choice has a little affect on the execution time. As n grows, the choice of the algorithm and data structure selection becomes significant. While n does not grow arbitrarily large, it is convenient to look at the limit of functions as n goes to infinity. Observe that for $f(n) = 6n \log_2 n$ and the function $g(n) = n^2$,

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0.$$

This limit captures the fact that the ratio of $f(n)$ to $g(n)$ decreases to an arbitrarily small value as n increases. When this occurs we say that $f(n)$ is **asymptotically slower growing** than $g(n)$, and

that $g(n)$ is **asymptotically faster growing** than $f(n)$. When

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$$

for any constant c , we say that $f(n)$ and $g(n)$ are **asymptotically equal**. While, one of these two functions may be smaller than the other, their ratio is bounded.

Asymptotic notation provides a way to describe the relationship between the asymptotic growth of two functions. The following asymptotic notation is used in this book, where $f(n)$ and $g(n)$ are non-negative functions.

Big-Oh (O): We say that $f(n) = O(g(n))$ when there exists some constant n_0 and some constant c such that $\forall n \geq n_0, f(n) \leq cg(n)$. We often talk about the constant c as the constant “hidden within the asymptotic notation.” An equivalent definition is that $f(n) = O(g(n))$ when

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$$

is either a constant or 0. We write $f(n) \leq_{lim} g(n)$ to denote that $f(n)$ grows asymptotically slower than or equal to $g(n)$.

Big-Omega (Ω): We say that $f(n) = \Omega(g(n))$ when that there exists some constant n_0 and some constant c such that $\forall n \geq n_0, f(n) \geq cg(n)$. Equivalently, $f(n) = \Omega(g(n))$ when

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$$

is either a constant or ∞ . We write $f(n) \geq_{lim} g(n)$ to denote that $f(n)$ grows asymptotically greater than or equal to $g(n)$.

Big-Theta (Θ): We say that $f(n) = \Theta(g(n))$ when $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. Equivalently, $f(n) = \Theta(g(n))$ when

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$$

for any constant c . We write $f(n) =_{lim} g(n)$ to denote that $f(n)$ and $g(n)$ are asymptotically equal.

Little-Oh (o): We say that $f(n) = o(g(n))$ when

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0.$$

We write $f(n) <_{lim} g(n)$ to denote that $f(n)$ grows asymptotically slower than $g(n)$.

Little-Omega (ω): We say that $f(n) = \omega(g(n))$ when

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty.$$

We write $f(n) >_{lim} g(n)$ to denote that $f(n)$ grows asymptotically faster than $g(n)$.

Observe that if $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then by transitivity it follows that $f(n) = O(h(n))$. Intuitively, this is like saying that if $f(n) \leq g(n)$ and $g(n) \leq h(n)$, then $f(n) \leq h(n)$. The difference, is that the asymptotic notation is based on the behavior as n goes to infinity. For all of the asymptotic notation defined above, the transitive property holds. Also, if $f(n) = O(g(n))$, then $g(n) = \Omega(f(n))$. In other words, if $f(n) \leq_{lim} g(n)$, then $g(n) \geq_{lim} f(n)$.

The importance of asymptotic notation is that when $f(n) = o(g(n))$, then generally an algorithm with time complexity $f(n)$ is going to be preferred to one of time complexity $g(n)$. The one caveat is that this notation would allow a situation where $f(n) = o(g(n))$, but the **crossing point** where $g(n)$ remains greater than $f(n)$ could be at a value of n greater than the size of any reasonable input. However, this situation rarely occurs.

$\log_2 n$	$\sqrt{n}$	n	$n \log_2 n$	n^2	n^3
0	1.0	1	0	1	1
1	1.4	2	2	4	8
2	2.0	4	8	16	64
3	2.8	8	24	64	512
4	4.0	16	64	256	4,096
5	5.7	32	160	1,024	32,768
6	8.0	64	384	4,096	262,144
7	11.3	128	896	16,384	2,097,152
8	16.0	256	2,048	65,536	16,777,216
9	22.6	512	4,608	262,144	134,217,728
10	32.0	1,024	10,240	1,048,576	1,073,741,824
11	45.3	2,048	22,528	4,194,304	8,589,934,592
12	64.0	4,096	49,152	16,777,216	68,719,476,736

Figure B.2
A comparison of various time complexity functions for sample values of n

As a rule of thumb, selecting an algorithm with the lowest asymptotic time complexity is the right choice. The following commonly occurring asymptotic time complexities go from the slowest to fastest growing. Table B.2 shows values of these functions, for sample values of n .

- $\Theta(1)$ is known as **constant** asymptotic time complexity.
- $\Theta(\log n)$ is known as **logarithmic** asymptotic time complexity.
- $\Theta(\sqrt{n})$
- $\Theta(n)$ is known as **linear** asymptotic time complexity.
- $\Theta(n \log n)$
- $\Theta(n^2)$ is known as **quadratic** asymptotic time complexity.
- $\Theta(n^3)$ is known as **cubic** asymptotic time complexity.

Observe that $\Theta(\log n)$ does not include the base of the logarithm. The reason for this decision, is that for any constants a and b ,

$$\lim_{n \rightarrow \infty} \frac{\log_a n}{\log_b n} = \lim_{n \rightarrow \infty} \frac{\ln n \ln b}{\ln a \ln n} = \lim_{n \rightarrow \infty} \frac{\ln b}{\ln a}$$

which is a constant. So $\log_a n = \Theta(\log_b n)$. Table B.2 uses a base of 2.

A **polynomial-time algorithm** is defined as any algorithm for which the asymptotic time complexity is $\Theta(n^c)$ for some constant c , and an **exponential-time algorithm** is defined as any algorithm for which the asymptotic time complexity is $\Theta(a^n)$ for some constant a .

B.3 Space Complexity

Although time complexity is typically the most important consideration, **space complexity** is also important as a measure of space usage as a function of the input size. Asymptotic notation can be used to describe the space complexity in the same way that it is used to describe the time complexity. All data structures maintain relatively any application-defined objects. Although the objects themselves often account for a significant fraction of the overall space usage, we define the **space complexity** of a data structure as the space usage of the data structure itself. We do not include the space used by application-defined objects that the data structure may reference. That is, we are interested only in the space overhead introduced by the data structure itself.

As an example to illustrate how we compute the space complexity, consider a singly-linked list (Chapter 15). Let $e_1, \dots, e_n$ be the elements that are maintained by the list. The list is composed of list item objects. The i^{th} list item has a reference to e_i , and a reference to the list item for e_{i+1} (which is *null* for $i = n$). There is also some space usage added by Java's run-time system: every object includes a reference to its type, and any instance of a non-static inner class includes a reference to the containing instance.

Thus, if the list item class is static (as in our singly-linked list) then each list item has space for 3 references, for a total space usage of roughly $3n$. If the list item inner class was not declared as a static class then the total space usage would be roughly $4n$. In either case, the space usage is $\Theta(n)$.

Asymptotic time complexities for methods can vary dramatically between different data structures. Thus, computing the asymptotic time complexity generally provides sufficient guidance to select an appropriate data structure. However, the space complexity is usually linear in the size of the collection so we generally state the time complexity in the form $c \cdot n$ for some constant c to provide a basis for comparison.

B.4 Expected Time Complexity

When an algorithm is randomized or when the time complexity is measured based on a random input, the number of statements $T_A(x)$ executed by algorithm A on input x is a **random variable**, so the mathematical definition of expected value can be applied. Namely if T is the random variable for the number of statements executed then

$$E[T] = \sum_t t \cdot \text{Probability}(T = t)$$

where the summation goes over all possible values t for T . More generally, when the probability distribution is continuous, the summation is replaced by an integral.

We consider two different forms of expected time complexity in this book. For a **randomized algorithm**, where a random decision is made during the execution of the algorithm (e.g., quicksort), no assumptions are made about the input. Let $T_A(x)$ be the number of statements executed by algorithm A on input x . We say that a randomized algorithm has **expected time complexity** $f(n)$ when for all inputs x where $|x| \leq n$, $E[T_A(x)] \leq f(n)$ where the expectation is taken with respect to the randomization of the algorithm.

The other case is when the expectation is taken with respect to a distribution (e.g., a uniform distribution) defined over the possible inputs. In this case we say that

$$E[T_A(x)] = \sum_t t \cdot \text{Probability}(f(x) = t)$$

where the expectation is taken over the probability distribution of the possible inputs.

B.5 Amortized Analysis

There are many data structures in which a condition can trigger a major restructuring of the data structure. Examples of this in this book include the union-find data structure (Section 6.3), dynamic arrays (Chapter 13), splay trees (Chapter 35), pairing heaps (Chapter 27), and Fibonacci heaps (Chapter 28).

Often, any reorganizing operation is guaranteed to be preceded by a sequence of operations that are very efficient, and can each be “charged” a small amount to help pay for the more expensive future cost. As a simple example, in a dynamic array, we double the size of the array each time it fills up. Because of this, occasionally an insertion will require $O(n)$ time. It can be proven that a length n sequence of insertions to the end of the array can always be done in $O(n)$ time, so the amortized time per operation has amortized constant cost, since the average cost of each insert $O(n)/n = O(1)$.

In general, the **amortized cost** of a sequence of operations is the average cost per operation of the worst-case cost of the sequence of operations. There is an important distinction between amortized analysis and expected-case analysis. The expected time complexity is an expected value over the random choices made by the algorithm, or an assumption about the distribution of the inputs. The actual time complexity might be larger than the expected value. The interpretation for an expected time complexity of $T(n)$ is that the average time complexity over an infinity sequence of executions of the algorithm approaches $T(n)$. Some executions will be faster, and other will be slower. In contrast, the amortized complexity provides a worst-case guarantee. The actual cost for a method

can be higher than the amortized cost, but the sum of the amortized cost of all methods is a worst-case bound. There are several techniques used in amortized analysis:

Aggregate Analysis determines the upper bound $T(n)$ on the total cost of any sequence of n operations. The amortized cost is $T(n)/n$.

Accounting Method determines an individual cost for each method some of which is used to account for the cost of the method, and the rest is saved to pay for the cost of any reorganization that must be performed by later methods. No method can be performed if there is not surplus available to pay for its cost.

Potential Method provides a way to define the surplus available as a function of the data structure. It provides the most general form of analysis.

We use the simple example of a sequence of insert operations to the end of a dynamic array. Whenever the array is full, its size is doubled and all elements are copied into the new array. For this analysis, we assume that there is a cost of 1 (representing constant cost) to place an element at the end of the array when the array is not full. We treat the cost to resize a full array of x elements to a half-full array of $2x$ elements as x since x elements must be copied to the new array*. We now use each of the above techniques to compute the amortized cost for inserting an element into the dynamic array. For ease of exposition, we assume that $n = 2^x + c$ elements are inserted into an array within an initial size of 2 where $1 \leq c \leq 2^x$. So the final array size is 2^{x+1} .

Aggregate Analysis: The total cost is n plus the cost for resizing. For $i = 1, \dots, x$, the array is resized from 2^i to 2^{i+1} with a cost of 2^i . Thus the total resizing cost is

$$\sum_{i=1}^x 2^i = 2^{x+1} - 2 \leq 2^{x+1} \leq 2n.$$

Thus the total cost for n insertions is at most $3n$. So the amortized cost of each operation is at most $3n/n = 3$.

Accounting Method: Each insertion is charged \$3. The actual insertion is charged \$1, and the remaining \$2 are “deposited in the bank” for later use. We now argue that whenever the array is resized from x to $2x$, that there will be at least x in the bank that can be used. The array is only resized from x to $2x$, after $x/2$ new elements are inserted. Each of the insertions deposited \$2 in the bank. Thus there is at least $2(x/2) = x$ in the bank. This can be used to pay for the resizing. Thus, at any point during the execution, there is always enough money in the bank to pay the resizing cost. So, clearly the total cost is at most $3n$. Observe that the amortized cost is the cost charged to each element.

Potential Method: To use the potential method, one defines a potential function Φ , which is a function of the structure, and is required to be non-negative. Intuitively, Φ captures the amount that has been “invested” in the data structure to cover later reorganization.

To illustrate this technique, we consider any sequence of operations that add to the end of the array or remove from the end of the array. When the array is full its capacity is doubled, and when the array is 1/4 utilized, its capacity is halved. The cost to change the array size from $2x$ to x is the cost of initializing the new array of size x , plus the cost of copying the $x/2$ elements from the original array into the new array. So, the total cost of reducing the array

*This analysis could be easily modified to also include the cost of initializing the $2n$ elements in the new array to *null* (as done by the Java runtime system). The asymptotic cost would not change.

size from $2x$ to x is $3x/2$. As in the aggregate analysis, we separate the cost of resizing from the cost of the actual add or remove operations.

Let a be the array, the potential Φ is defined as:

$$\Phi = \begin{cases} 2n - a.length & \text{if } n \geq a.length/2 \\ a.length/2 - n & \text{if } n \leq a.length/2 \end{cases}$$

where n is the number of elements held in the array. Intuitively, Φ keeps track of the “credits” available from insertions and removals that can be saved to “pay” for occasional resizing operations. When the array is at least half full, we have two credits for each element in the second half, which can be used to pay for growing the array. When the array is less than half full, we have one credit for each empty slot in the first half, which can be used to pay for shrinking the array. Observe that $\Phi = 0$ when $n = a.length/2$.

The table below summarizes the analysis to compute the change in potential when adding an element or removing an element, for doubling the array capacity, and for halving the array capacity. Let s be the array size when the method is called.

method	conditions	old Φ	new Φ	$\Delta\Phi$
add	$n \geq s/2$	$2n - s$	$2(n+1) - s$	+2
	$n < s/2$	$s/2 - n$	$s/2 - (n+1)$	+1
remove	$n > s/2$	$2n - s$	$2(n-1) - s$	-2
	$n \leq s/2$	$s/2 - n$	$s/2 - (n-1)$	-1
double array size	$n = s$	$2n - s = n$	$2n - 2n = 0$	-n
half array size	$n = s/4$	$s/2 - n = n$	$s/2 - n = 0$	-n

The potential cost for each operation is an upper bound the actual cost plus the change in potential (new Φ - old Φ). Since the actual cost to add an element is +1, and the maximum increase in potential is +2, the amortized cost for adding an element is +3. Similarly, the actual cost to remove an element is +1, and the maximum value for the change in potential is -1. So the amortized cost to remove an element is 0.

Since the actual cost to double the array from n to $2n$ is n , and the potential is reduced by n , the amortized cost is 0. Likewise, the actual cost to half the array capacity from $4n$ to $2n$ is n and the potential is reduced by n , so the amortized cost of 0.

Thus the amortized cost to add an element (including the resizing) is +3, and the amortized cost to remove an element (including the resizing) is 0. Thus any sequence of n operations has cost at most $3n$.

B.6 Solving Recurrence Equations with the Master Method

There are several variations of the master method. The variation we present will solve any recurrence equation of the form:

$$\begin{aligned} T(n) &= \Theta(1) \text{ for all } n \leq c \\ T(n) &= aT(n/b) + f(n) \text{ for all } n > c \end{aligned}$$

where $f(n) = \Theta(n^\ell(\log n)^k)$ for constants, $c \geq 0$, $a \geq 1$, $b > 1$, $\ell \geq 0$, and $k \geq 0$. Such recurrences arise naturally in the time complexity analysis for recursive algorithms that divides the problem into a subproblems of size roughly n/b . These a subproblems are solved recursively, and then the a solutions are combined to obtain the solution to the original problem. The function $f(n)$ is the sum of the cost to divide the original problem into subproblems and to combine the subproblem solutions. For an example of such an algorithm see merge sort 11.4.2.

For ease of exposition, throughout the remainder of this section we assume n is a power of b . The asymptotic value holds without this assumption since the cost is upper bounded by that analysis when n is rounded up to the nearest power of b (which can be no more than b times larger than its value). All recurrences are placed into one of three cases based on the relationship between ℓ and $\log_b a$. The time to split and combine is $O(n^\ell)$ and the number of times the termination condition is reached, when

For recurrences of the form $T(n) = aT(n/b) + f(n)$, when the termination condition is for $n = 1$, there are exactly $a^{\log_b n} = n^{\log_b a}$ applications of the base case of the recurrence. So $\Theta(n^{\log_b a})$ time is spent at the base of the recursive calls. Intuitively, when $\ell < \log_b a$, the number of calls to the termination condition is as much (asymptotically), as the total computation time for all the calls that do not reach the termination condition. When $\ell = \log_b a$, exactly $f(n)$ time is spent to solve all subproblems of any given size, and there are $\log_b n$ different levels of recursion (and hence subproblem sizes). Thus the total cost is $f(n) \cdot \log_b n$. Finally, when $\ell > \log_b a$, the computation at the top-level call is as much (asymptotically) as all the rest of the computation.

Case 1: For $\ell < \log_b a$, $T(n) = \Theta(n^{\log_b a})$

Case 2: For $\ell = \log_b a$, $T(n) = \Theta(f(n) \log n) = \Theta(n^{\log_b a}(\log n)^{k+1})$

Case 3: For $\ell > \log_b a$, $T(n) = \Theta(f(n)) = \Theta(n^\ell(\log n)^k)$

The advantage of this formulation is that it can apply for any value of $k \geq 0$, and it is easy to apply. We now illustrate each case using an example. To provide some intuition behind these cases, by using a **recurrence tree** in which each recursive call made, with its size is shown, one can derive that for a the recurrence $T(n) = aT(n/b) + f(n)$, for $n \geq 1$ a power of b with termination at $T(1)$, the exact solution is given by the summation:

$$T(n) = \left(\sum_{i=0}^{(\log_b n)-1} a^i f\left(\frac{n}{b^i}\right) \right) + T(1)a^{\log_b n}$$

since at level i of the recursion tree, where $i = 0$ at the root and $i = \log_b n$ at the leaves, there are a^i nodes. Except for the leaves $f(\frac{n}{b^i})$ time is spent, not counting the recursive calls, and $T(1)$ time is spent per leaf. We use the standard summation for the **arithmetic series** that

$$\sum_{i=1}^n i = n(n+1)/2,$$

and the standard summation for the **geometric series** that

$$\sum_{i=0}^r x^i = (x^{r+1} - 1)/(x - 1) = (1 - x^{r+1})/(1 - x)$$

for $x \neq 1$. Differentiating both sides of the summation for the geometric series with respect to x , and multiplying both sides by x , yields $\sum_{i=0}^r i \cdot x^i = (x^{r+1}(r(x-1) - 1) + x)/(x-1)^2$. So for $x = 2$, $\sum_{i=0}^r i \cdot 2^i = 2^{r+1}(r-1) + 2$.

To give some intuition behind the master method, for each case of the master method we compute the exact solution for the recurrence. We also include one example for which the master method does not apply.

Master Method: Case 1 Example. Here we consider $T(n) = 4T(n/2) + 3n \log_2 n$, $T(1) = 1$ for $n \geq 1$ a power of 2. Applying the above general form for the summation gives

$$\begin{aligned}
 T(n) &= \left(\sum_{i=0}^{(\log_2 n)-1} 4^i \left(3 \cdot \frac{n}{2^i} \log_2 \frac{n}{2^i} \right) \right) + T(1) \cdot 4^{\log_2 n} \\
 &= \left(3n \sum_{i=0}^{(\log_2 n)-1} 2^i (\log_2 n - i) \right) + n^2 \\
 &= \left(3n \log_2 n \sum_{i=0}^{(\log_2 n)-1} 2^i \right) - \left(3n \sum_{i=0}^{(\log_2 n)-1} i \cdot 2^i \right) + n^2 \\
 &= 3n \log_2 n (2^{\log_2 n} - 1) - 3n (2^{\log_2 n} (\log_2 n - 2) + 2) + n^2 \\
 &= 3n^2 \log_2 n - 3n \log_2 n - 3n^2 \log_2 n + 6n^2 - 6n + n^2 \\
 &= 7n^2 - 3n \log_2 n - 6n = \Theta(n^2)
 \end{aligned}$$

Master Method: Case 2 Example. Here we consider $T(n) = 2T(n/4) + \sqrt{n}$, $T(1) = 1$ for $n \geq 1$ a power of 4. Applying the above general form for the summation we have

$$\begin{aligned}
 T(n) &= \left(\sum_{i=0}^{(\log_4 n)-1} 2^i \sqrt{n/4^i} \right) + T(1) \cdot 2^{\log_4 n} \\
 &= \left(\sum_{i=0}^{(\log_4 n)-1} \sqrt{n} \right) + \sqrt{n} = \sqrt{n} \log_4 n + \sqrt{n} = \Theta(\sqrt{n})
 \end{aligned}$$

Master Method: Case 3 Example. Here we consider $T(n) = 2T(n/2) + 3n^2$, $T(1) = 2$ for $n \geq 1$ a power of 2. Applying the above general form for the summation we have

$$\begin{aligned}
 T(n) &= \left(\sum_{i=0}^{(\log_2 n)-1} 2^i \cdot 3 \left(\frac{n}{2^i} \right)^2 \right) + T(1) \cdot 2^{\log_2 n} \\
 &= \left(3n^2 \sum_{i=0}^{(\log_2 n)-1} 2^i / 4^i \right) + 2n \\
 &= \left(3n^2 \sum_{i=0}^{(\log_2 n)-1} (1/2)^i \right) + 2n \\
 &= 3n^2 \left(\frac{1 - (1/2)^{\log_2 n}}{1 - 1/2} \right) + 2n \\
 &= 6n^2 \left(1 - \frac{1}{n} \right) + 2n \\
 &= 6n^2 - 4n = \Theta(n^2)
 \end{aligned}$$

A Recurrence Relation not of the Master Method Form. Here we consider the recurrence equation $T(n) = T(n-2) + 2n$, $T(0) = 1$ for $n \geq 0$ an even integer. In this recurrence tree, at the i th level the problem will be of size $n - 2i$ with the root having $i = 0$ and the leaf having $i = n/2$ (so

$n - 2i = 0$). For the internal nodes, the cost at level i is $2i$ and for the leaf the cost is 1. Finally, there is just one node at each level of the tree. So we have

$$\begin{aligned}
 T(n) &= \left(\sum_{i=0}^{(n/2)-1} 1 \cdot 2(n - 2i) \right) + T(0) \\
 &= \left(\sum_{i=0}^{(n/2)-1} 2n - 4i \right) + 1 \\
 &= 2n \cdot (n/2) - \left(\sum_{i=0}^{(n/2)-1} 4i \right) + 1 \\
 &= n^2 - \left(4 \cdot \frac{1}{2} \left(\frac{n}{2} - 1 \right) \frac{n}{2} \right) + 1 \\
 &= n^2 - 2 \left(\left(\frac{n}{2} - 1 \right) \frac{n}{2} \right) + 1 \\
 &= \frac{n^2}{2} + n + 1 = \Theta(n^2)
 \end{aligned}$$

Appendix C

Design Patterns Illustrated in This Book

Programmers have noticed many useful patterns that occur naturally in the design of object-oriented software. In this section, we describe patterns that appear throughout this book. For additional examples of design patterns see *Design Patterns: Elements of Reusable Object-Oriented Software* [71]. Another good book that includes discussion of many design patterns is *Program Development in Java: Abstraction, Specification, and Object-Oriented Design* [105].

C.1 Abstract Factory

Provide an interface for creating families of related or dependent objects without specifying their concrete classes. [71]

We use this design pattern to provide functionality similar to that of a virtual constructor. Specifically, a bucket factory (Section 50.3) is used by a tagged bucket collection (Chapter 50) to create a new bucket for each newly added tag. Using this design pattern provides the user flexibility to create a bucket collection that uses the desired collection type for the buckets.

C.2 Adaptor

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. [71]

One use of the adaptor design patterns is to let classes work together. For example, each Patricia trie node (Chapter 44) must serve as both an internal node and leaf node. To achieve this task, the Patricia trie node must inherit the functionality of both of these classes. However, Java does not support multiple inheritance. We resolve this difficulty by using the adaptor design pattern to enable the Patricia trie node to share behaviors of the compressed trie internal nodes and leaf nodes.

The adaptor design pattern is also used make an existing object adapt to a different interface. The buffer (Chapter 17), queue (Chapter 18), and stack (Chapter 19) provide an example use of the adaptor design pattern to hide some of the functionality of the positional collection to create a more specialized interface.

C.3 Bridge

Separates the implementation hierarchy for a type from its subtype hierarchy. [105].

Decouple an abstraction from its implementation so that the two can vary independently [71].

We use the bridge pattern in the `AbstractGraph` class (Chapter 53) and the `AbstractWeightedGraph` class (Chapter 57) by having the graph representation classes be a bridge between the underlying representation and the application usage. The *GraphRepresentation* interface (Section 52.5) defines the methods that must be supported by any graph representation, such as the adjacency list and adjacency matrix representations. By using the bridge design pattern, we allow different types of graphs (e.g., weighted and unweighted) to independently choose their internal representation from among the classes that implement *GraphRepresentation*.

We also use the bridge design pattern to enable the graph algorithm to be implemented entirely in terms of either the Graph ADT (Chapter 52) or `WeightedGraph` ADT (Chapter 56). This allows the presentation and implementation of the graph algorithms to be expressed generally, without reference to a specific underlying graph representation.

C.4 Builder

Separate the construction of a complex object from its representation so that the same construction process can create different representations. [71]

The buffer (Chapter 17), queue (Chapter 18), and stack (Chapter 19) use the builder design pattern. In particular, these data structures support a polymorphic construction for a wrapped object that adheres to a general interface (in this case, a positional collection). For example, we consider bounded and unbounded buffers as well as tracked and untracked versions of each. To create the wrapped positional collection, the constructor includes logic to select an appropriate positional collection to support the desired buffer properties. The allocation of the object with *new* plays the builder role, and the constructor plays the director role by selecting automatically among alternative internal representations.

C.5 Command

Encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. [71]

We use this design pattern for the purpose of deferring invocation of a method so that an object type can serve multiple roles. In particular, a common use of the abstract factory design pattern is the *Comparator* interface (Section 5.2.2). This interface allows the ordering among elements to be defined independently from the data structure or algorithm that depends on that ordering.

The *Digitizer* interface (Section 5.3) and the *Bucketizer* interface (Section 5.4) are additional examples of this use of the command design pattern. These interfaces allow an algorithm that uses a digitizer (e.g., radix sort) or bucketizer (e.g., bucket sort), or a data structure that uses a digitizer

(e.g., trie, compact trie, compressed trie, Patricia trie, and ternary search trie) to be implemented independently from the digitizer or bucketizer.

C.6 Composite

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. [71]

One way in which we use the composite design pattern is to combine two data types within a single data structure. For example, in the trie implementations (see Chapter 40) each child reference is to a trie node that could either be a trie node or a trie leaf node. This design avoids the need to have the implementation of the `TrieLeafNode` extend the `TrieNode`. The advantage of using this approach is that the leaf node does not waste space for unnecessary instance variables such as the child references.

All of our tree data structure implementations (`AbstractSearchTree`, Chapter 31) can be viewed as composites. Each tree node may hold one or more elements, but it also represents an entire subtree. Operations such as rotation in a balanced binary search tree (Chapter 33) take advantage of this view by treating subtrees and individual nodes uniformly.

C.7 Decorator

Interposes an object between using code and the original object in order to extend the behavior of the original object. The interposed object will belong to a subtype of the type of the original object. [105]

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. [71]

The separation of the `KDTreeImpl` class (Section 47.5) from the `KDTree` class illustrates a use of this design pattern. While it may seem natural for the `KDTree` class to directly implement the data structure, the dichotomy is necessary to resolve two competing design objectives. On one hand, the structure of a k -d tree is similar to a binary search tree, so it is desirable to extend the `BinarySearchTree` class. On the other hand, a k -d tree is technically not a binary search tree, and it would not make sense for a k -d tree to provide methods like *min*, *max*, *predecessor*, and *successor* because these are not uniquely defined unless the dimension of comparison is specified. The decorator allows the desired structure to be inherited without exposing the binary search methods. Additional methods (e.g., *withinBounds* to perform an orthogonal range query) are decorators on the core.

C.8 Facade

Provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. [71]

We use a facade wrapper to build tagged collection (Chapter 49) from the appropriate collection data structure.

C.9 Factory Method

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. [71]

We use this design pattern throughout this book to construct more specific object instances of an inner class to increase the amount of code reuse. For example, by introducing the factory method *newListItem* to the singly linked list implementation (Chapter 15), all of the public positional collection methods can be inherited by the doubly linked list implementation (Chapter 16). This reuse of code is enabled by overriding the singly linked list *List*Item class by a doubly linked list *List*Item class for which the methods are extended to also adjust the previous reference. When a method such as *add* creates a new list item it uses the *newListItem* method. The only other methods from the *SinglyLinkedList* class that are modified are those that can be implemented more efficiently by using the previous references.

The trie (Chapter 41), compact trie (Chapter 42), compressed trie (Chapter 43), Patricia trie (Chapter 44), and ternary search trie (Chapter 45) use the factory method design pattern to create the appropriate internal node and leaf node types.

Our test cases also use factory methods extensively. To support inheriting tests to be executed on subtypes, each test case calls a factory method to instantiate the object under test. By extending the test class and overriding the factory method, the same test suite can be applied to subtypes without duplication of code.

C.10 Flyweight

Use sharing to support large numbers of fine-grained objects efficiently. [71]

A binary search tree (Chapter 32) with n nodes has $n+1$ frontier nodes. A common representation is to represent each frontier node as a *null* reference. However, with access to the frontier node where an unsuccessful search ends, and the ability to access the parent of that frontier node, insertion can be performed in constant time. As discussed in Section 32.1, instead of creating $n+1$ frontier nodes, we apply the flyweight pattern to create two shared frontier nodes, *FRONTIER_L* and *FRONTIER_R*, to greatly reduce the space usage of our implementation. Although the flyweight pattern more typically involves creating new objects on demand when an instance is not yet available for sharing, this simplified application of the flyweight pattern creates both frontier nodes in advance and then uses sharing exclusively.

C.11 Iterator

Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. [71]

Iterators (see Section 5.7) are supported for all implementations in this book. In fact, the *Locator* interface (Section 5.8) is an extension of the *Iterator* interface. A locator can either mark a position

within a collection (sometimes called the **position design pattern** [76]) or track a specific element (sometimes called the **locator design pattern** [76]).

C.12 Leasing

Simplifies a resource management by specifying how resource users can get access to a resource from a resource provider for a pre-defined period of time.[89]

We use a variation of this design pattern in which the borrower explicitly releases the resource rather than leasing it for a bounded period of time. In particular, the object pool discussed in Section 5.5 is used by the Trie (Chapter 41) class for the objects that hold information computed during search.

C.13 Proxy

Provides a surrogate or placeholder for another object to control access to it. [71]

The union-find data structure (page 76) illustrates the **proxy design pattern** by having the representative element of each component serve as a surrogate for the entire component. The leftist heap data structure (Chapter 26) illustrates how the proxy design pattern allows two priority queues, each of which might have some tracked element and active iterators, to be merged by introducing a union-find node to serve as the proxy that references the representative leftist heap among a set that have been merged into a single leftist heap. Section 6.8 discusses this use of the proxy design patterns and the union-find data structure in more depth.

Another common use of the proxy design pattern throughout this book is the redirect chain that is first introduced in the singly linked list data structure (Chapter 15). In particular, whenever a tracker t is at an element that is removed, the redirect chain from t ends at the element in the collection that serves as the proxy for the tracker, in that the current tracker location is either immediately before (or after) its proxy.

C.14 Singleton

Ensures a class has only one instance, and provides a global point of access to it. [71]

The objects *EMPTY* and *DELETED* (Section 5.2.1) are examples uses of this design pattern which ensures that each class has one instance and provides a global access point. We use this design pattern in many places to reduce space. The singleton *REMOVED* defined in the *SinglyLinkedList* class (Chapter 15) is another example use of the singleton design pattern.

C.15 Strategy

Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithms vary independently from clients that use them. [71]

Comparators (Section 5.2.2), digitizers (Section 5.3) and bucketizers (Section 5.4) illustrate an example use of this design pattern to decouple the mechanism to define an order of the elements from data structures and algorithms that rely on this ordering. For example, the sorting algorithms presented as part of the Array class (Chapter 11) and SinglyLinkedList class (Chapter 15) data structures use the strategy design pattern for the purpose of replacing the comparator used to sort. Consider an application that needs to sort a collection of two-dimensional points. The points could be ordered based on the x -coordinate, based on the y -coordinate, or perhaps based on the distance from the origin. Yet the design of an algorithm to sort the points is independent of the way in which the comparator is defined, requiring a mechanism to provide the desired comparator.

The use of a comparator in this way is sometimes defined as the **comparator design pattern** [76].

C.16 Template Method

Defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. [71]

This design pattern is heavily used throughout this book. We mention just a few examples here. In the AbstractCollection class (Chapter 8), many of the public methods are implemented in terms of abstract methods like *add*. This avoids duplication of code by enabling properties such as SIZE to be maintained in the abstract collection implementation.

The array data structure (Chapter 11) uses the template method design pattern to translate between the user position and the underlying index. By separating the *getIndex*, *getPosition*, *nextIndex* and *prevIndex* methods from the rest of the computation, the methods that rely on the translation can be inherited in their entirety by the CircularArray class (Chapter 12) by just replacing the few methods that specifically perform this translation.

C.17 Visitor

Represents an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates. [71]

A design pattern in which a traversal of a composite tree is accomplished by passing a visitor object to each node of the tree. There is a visitor class per kind of traversal. [105]

All of our collections implementations accept visitors. This design pattern is discussed in more depth in Section 5.10.

References

- [1] Wilhelm Ackermann. Zum hilbertschen aufbau der reellen zahlen. *Mathematische Annalen*, 99:118–133, 1928.
- [2] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.
- [3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.
- [4] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.
- [5] Ravindra K. Ahuja, Kurt Mehlhorn, James B. Orlin, and Robert E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37:213–223, 1990.
- [6] Ravindra K. Ahuja and James B. Orlin. A fast and simple algorithm for the maximum flow problem. *Operations Research*, 37(5):748–759, 1989.
- [7] Ravindra K. Ahuja, James B. Orlin, and Robert E. Tarjan. Improved time bounds for the maximum flow problem. *SIAM Journal on Computing*, 18(5):939–954, 1989.
- [8] M. Al-Suwaiyel and E Horowitz. Algorithms for trie compaction. *ACM Transactions on Database Systems*, 9(2):243–263, 1984.
- [9] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.
- [10] Stephen F. Altschul, Thomas L. Madden, Alejandro A. Schäffer, Jingui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman. Grapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, 1997.
- [11] Arne Andersson. Balanced search trees made simple. In *Proceedings of the Third Workshop on Algorithms and Data Structures*, number 709 in Lecture Notes in Computer Science, pages 60–71. Springer-Verlag, 1993.
- [12] Arvind Arasu, Junghoo Cho, Hector Garcia-Molina, Andreas Paepcke, and Sriram Raghavan. Searching the web. *ACM Transactions on Internet Technology*, 1(1):2–43, 2001.
- [13] Lars Arge, Mark de Berg, Herman J. Haverkort, and Ke Yi. The priority R-tree: a practically efficient and worst-case optimal R-tree. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 347–358, 2004.
- [14] Sunil Arya and David M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proceedings of the 4th ACM-SIAM Symposium on Discrete Algorithms*, pages 271–280, 1993.
- [15] Sunil Arya and David M. Mount. Approximate range searching. In *Symposium on Computational Geometry*, pages 172–181, 1995.
- [16] M. D. Atkinson, J.-R. Sack, B. Santoro, and T. Strothotte. Min-max heaps and generalized priority queues. *Communications of the ACM*, 29(10):996–1000, 1986.
- [17] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

- [18] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.
- [19] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 322–331, 1990.
- [20] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.
- [21] Jon Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.
- [22] Jon Bentley. Programming pearls: Writing correct programs. *Communications of the ACM*, 26(12):1040–1045, 1983.
- [23] Jon L. Bentley. Programming pearls: How to sort. *Communications of the ACM*, 27(4):287–291, 1984.
- [24] Jon L. Bentley and M. Douglas McIlroy. Engineering a sort function. *Software - Practice and Experience*, 23(11):1249–1265, 1993.
- [25] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms*, 1997.
- [26] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.
- [27] Otakar Borůvka. O jistém problému minimálním. *Práce Moravské Přírodovědecké Společnosti*, 3:37–58, 1926. In Czech.
- [28] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly and Associates, Inc., third edition, 2005.
- [29] Yuri Boykov and Gareth Funka-Lea. Graph cuts and efficient N-D image segmentation. *International Journal of Computer Vision*, 70(2):109–131, 2006.
- [30] Yuri Boykov and Vladimir Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE transactions on Pattern Analysis and Machine Intelligence*, 26(9):1124–1137, September 2004.
- [31] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.
- [32] Andrej Brodnik, Svante Carlsson, Erik D. Demaine, J. Ian Munro, and Robert Sedgewick. Resizable arrays in optimal time and space. In *Workshop on Algorithms and Data Structures*, pages 37–48, 1999.
- [33] Mark R. Brown. *The Analysis of a Practical and Nearly Optimal Priority Queue*. PhD thesis, Computer Science Department, Stanford University, 1977. Technical Report STAN-CS-77-600.
- [34] Mark R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal on Computing*, 7(3):298–319, 1978.
- [35] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2003.
- [36] R. Creighton Buck. Mathematical induction and recursive definitions. *American Mathematical Monthly*, 70:128–135, 1963.

- [37] Svante Carlsson. The deapa double-ended heap to implement double-ended priority queues. *Information Processing Letters*, 26(1):33–36, 1987.
- [38] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Third Symposium on Operating Systems Design and Implementation*, February 1999.
- [39] Bernard Chazelle. A minimum spanning tree algorithm with inverse-Ackermann type complexity. *Journal of the ACM*, 47(6):1028–1047, 2000.
- [40] Julien Clément, Philippe Flajolet, and Brigitte Vallée. The analysis of hybrid trie structures. In *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms*, pages 531–539, 1998.
- [41] E. G. Coffman, Jr. and J. Eve. File structures using hashing functions. *Communications of the ACM*, 13(7):427–432, 1970.
- [42] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2001.
- [43] Clark A. Crane. Linear lists and priority queues as balanced binary trees. Technical Report CS-72-259, Computer Science Department, Stanford University, 1972.
- [44] Luc Devroye. A limit theory for random skip lists. *The Annals of Applied Probability*, 2(3):597–609, 1992.
- [45] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.
- [46] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
- [47] E. A. Dinic. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Mathematics Doklady*, 11(5):1277–1280, 1970.
- [48] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.
- [49] H. Edelsbrunner. A note on dynamic range searching. *Bulletin of the EATCS*, 15:34–40, 1981.
- [50] Jack Edmonds and Richard M. Karp. Theoretical improvements in the algorithmic efficiency for network flow problems. *Journal of the ACM*, 19:248–264, 1972.
- [51] Shimon Even. *Graph Algorithms*. Computer Science Press, 1979.
- [52] William Feller. *An Introduction to Probability Theory and Its Applications*. John Wiley & Sons, third edition, 1968.
- [53] Raphael Finkel and Jon L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974.
- [54] Robert W. Floyd. Algorithm 97 (SHORTEST PATH). *Communications of the ACM*, 5(6):345, 1962.
- [55] Robert W. Floyd. Algorithm 245 (TREESORT). *Communications of the ACM*, 7:701, 1964.
- [56] Robert W. Floyd. Assigning meaning to programs. *Mathematical aspects of computer science*, pages 19–32, 1967. XIX American Mathematical Society.

- [57] Robert W. Floyd and Ronald L. Rivest. Expected time bounds for selection. *Communications of the ACM*, 18(3):165–172, 1975.
- [58] Lestor R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.
- [59] Lestor R. Ford, Jr. and Selmer M. Johnson. A tournament problem. *The American Mathematical Monthly*, 66:387–389, 1959.
- [60] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.
- [61] Michael L. Fredman. On the efficiency of pairing heaps and related data structures. *Journal of the ACM*, 46(4):473–501, 1999.
- [62] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.
- [63] Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, 1989.
- [64] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: a new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.
- [65] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.
- [66] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47:424–436, 1993.
- [67] Jerome H. Freidman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, 1977.
- [68] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. In *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques*, pages 124–133, 1980.
- [69] Harold N. Gabow and Robert E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.
- [70] Harold N. Gabow and Robert E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18(5):1013–1036, 1989.
- [71] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [72] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.
- [73] Andrew V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1987.
- [74] Andrew V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM Journal on Computing*, 24(3):494–504, 1995.
- [75] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM*, 35:921–940, 1988.
- [76] Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, 1998.

- [77] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1995.
- [78] D. M. Greig, B. T. Porteous, and A. H. Seheult. Exact maximum a posteriori estimation for binary images. *Journal of the Royal Statistical Society, Series B*, 51(2):271–279, 1989.
- [79] David Gries. *The Science of Programming*. Springer-Verlag, 1981.
- [80] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21. IEEE Computer Society, 1978.
- [81] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.
- [82] C. A. R. Hoare. Algorithm 63 (PARTITION) and algorithm 65 (FIND). *Communications of the ACM*, 4(7):321–322, 1961.
- [83] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.
- [84] John E. Hopcroft and Robert E. Tarjan. Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.
- [85] John E. Hopcroft and Jeffrey D. Ullman. Set merging algorithms. *SIAM Journal on Computing*, 2(4):294–303, 1973.
- [86] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, 1978.
- [87] Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed. *Fundamentals of Data Structures in C*. Computer Science Press, 1993.
- [88] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
- [89] Prashant Jain and Michael Kircher. Leasing. In *Proceedings of the 2000 Patterns Languages of Programming Conference*, 2000.
- [90] V. Jarník. O jistém problému minimálním. *Práce Moravské Přírodovědecké Společnosti*, 6:57–63, 1930. In Czech.
- [91] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, 1977.
- [92] David R. Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42(2):321–328, 1995.
- [93] David R. Karger, Daphne Koller, and Steven J. Phillips. Finding the hidden path: time bounds for all-pairs shortest paths. *SIAM Journal on Computing*, 22(6):1199–1217, 1993.
- [94] Peter Kirschenhofer and Helmut Prodinger. The path length of random skip lists. *Acta Inf.*, 31(9):775–792, 1994.
- [95] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.
- [96] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, 1968. Second edition, 1973.

- [97] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.
- [98] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.
- [99] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.
- [100] Amy N. Langville and Carl D. Meyer. *Google's PageRank and Beyond*. Princeton University Press, 2006.
- [101] Douglas Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, third edition, 2006.
- [102] C. Y. Lee. An algorithm for path connection and its applications. *IRE Transactions on Electronic Computers*, EC-10(3):346–365, 1961.
- [103] R. Lesuisse. Some lessons drawn from the history of the binary search algorithm. *The Computer Journal*, 26:154–163, 1983.
- [104] Andrew M. Liao. Three priority queue applications revisited. *Algorithmica*, 7(4):415–427, 1992.
- [105] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2001.
- [106] Conrado Martínez and Salvador Roura. Randomized binary search trees. *Journal of the ACM*, 45(2):288–323, 1998.
- [107] Jiří Matoušek. Efficient partition trees. In *Proceedings of the 7th Annual Symposium on Computational Geometry*, pages 1–9, 1991.
- [108] Jiří Matoušek. Geometric range searching. *ACM Computing Surveys*, 26(4):421–461, 1994.
- [109] Edward M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14:257–2769, 1985.
- [110] C. C. McGeoch. All pairs shortest paths and the essential subgraph. *Algorithmica*, 13(5):426–441, 1995.
- [111] Peter M. McIlroy, Keith Bostic, and M. Douglas McIlroy. Engineering radix sort. *Computing Systems*, 6(1):5–27, 1993.
- [112] Kurt Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms*. Springer-Verlag, 1984.
- [113] Kurt Mehlhorn. *Multidimensional Searching and Computational Geometry*, volume 3 of *Data Structures and Algorithms*. Springer-Verlag, 1984.
- [114] John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996.
- [115] Edward F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.
- [116] Bernard M. E. Moret and Henry D. Shapiro. An empirical assessment of algorithms for constructing a minimum spanning tree. In Nathaniel Dean and Gregory E. Shannon, editors, *Computational Support for Discrete Mathematics*, volume 15 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 99–118. American Mathematical Society, 1994.

- [117] Robert Morris. Scatter storage techniques. *Communications of the ACM*, 11(1):38–44, 1968.
- [118] Donald R. Morrison. Patricia: Practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.
- [119] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
- [120] Sajeeva L. Pallemulle, Ian Wehrman, and Kenneth J. Goldman. Byzantine fault tolerant execution of long-running distributed applications. In *18th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 528–534, 2006.
- [121] Thomas Papadakis, J. Ian Munro, and Patricio V. Poblete. Average search and update costs in skip lists. *BIT*, 32(2):316–332, 1992.
- [122] Willaim R. Pearson and David J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences of the United States of America*, 85(8):2444–2448, 1988.
- [123] W. W. Peterson. Addressing for random-access storage. *IBM Journal of Research and Development*, 1(2):130–146, 1957.
- [124] Seth Pettie. Towards a final analysis of pairing heaps. In *Proceedings of the 46th Annual Symposium on Foundations of Computer Science*, pages 174–183, 2005.
- [125] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.
- [126] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.
- [127] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.
- [128] Sebastien Roy and Ingemar J. Cox. A maximum-flow formulation of the n-camera stereo correspondence problem. In *Proceedings of the 6th International Conference on Computer Vision*, page 492, 1998.
- [129] Gerald Salton and Chris Buckley. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.
- [130] Hanan Samet. Deletion in two-dimensional quad trees. *Communications of the ACM*, 23(12):703–710, 1980.
- [131] Hanan Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, 1990.
- [132] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.
- [133] Herbert Schorr and W.M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Journal of the ACM*, 10(8):501–506, 1967.
- [134] Robert Sedgewick. Implementing quicksort programs. *Communications of the ACM*, 21(10):847–857, 1978.
- [135] Robert Sedgewick. *Algorithms*. Addison-Wesley, second edition, 1988.
- [136] Robert Sedgewick. *Algorithms in C, Parts 1–4*. Addison-Wesley Longman Publishing Co., Inc., 2002.

- [137] Raimund G. Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996.
- [138] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-tree: A dynamic index for multi-dimensional objects. In *The VLDB Journal*, pages 507–518, 1987.
- [139] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers and Mathematics with Applications*, 7:67–72, 1981.
- [140] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.
- [141] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting heaps. *SIAM Journal on Computing*, 15(1):52–69, 1986.
- [142] John Stasko and Jeff Vitter. Pairing heaps: experiments and analysis. *Communications of the ACM*, 30(3):234–249, 1987.
- [143] Robert E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.
- [144] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.
- [145] Robert E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2):110–127, 1979.
- [146] Robert E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.
- [147] Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, 1984.
- [148] Mikkel Thorup. Faster deterministic sorting and priority queues in linear space. In *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms*, pages 550–555, 1998.
- [149] Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46(3):362–394, 1999.
- [150] Mikkel Thorup. On RAM priority queues. *SIAM Journal on Computing*, 30(1):86–109, 2000.
- [151] Haraldur D. Thorvaldsson and Kenneth J. Goldman. Dynamic evolution in a survivable application infrastructure. In *18th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 503–509, 2006.
- [152] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, pages 75–84. IEEE Computer Society, 1975.
- [153] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.
- [154] Jean Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23:229–239, 1980.
- [155] Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.
- [156] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Addison-Wesley, 1994.
- [157] J. Williams. Algorithm 232 (HEAPSORT). *Communications of the ACM*, 7:347–348, 1964.
- [158] Derick Wood. *Data structures, algorithms, and performance*. Addison-Wesley, 1993.