

Neil Dale
Susan C. Lilly
John McCormick

Ada
Plus Data Structures
An Object-Based Approach

Ada Plus Data Structures

An Object Based Approach

Nell Dale

The University of Texas at Austin

Susan C. Lilly

IBM

John McCormick

State University of New York at Plattsburgh

D. C. Heath and Company
Lexington, Massachusetts Toronto

Disclaimer:

This netLibrary eBook does not include the ancillary media that was packaged with the original printed version of the book.

Address editorial correspondence to:

D. C. Heath and Company
125 Spring Street
Lexington, MA 02173

Acquisitions: Walter Cunningham

Development: Karen H. Jolie

Editorial Production: Janice Molloy

Cover design: Jan Shapiro

Production Coordination: Charles Dutton

Cover: Lois Ellen Frank/Westlight

Copyright © 1996 by D. C. Heath and Company.

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopy, recording, or any information storage or retrieval system, without permission in writing from the publisher.

Published simultaneously in Canada.

Printed in the United States of America.

International Standard Book Number: 0-669-29265-6

Library of Congress Catalog Card Number: 95-68037

10 9 8 7 6 5 4 3 2 1

*This book is dedicated to you,
and to all of our students for whom it was begun and
without whom it could never have been completed.*

N. D. S. L. J. M.

To Naomi.

J. M.

Preface

In the past decade computer science has seen software engineering techniques put into practice to solve larger and more complex computing problems than ever before. By merging the study of data structures and algorithms with the related concept of data abstraction, *Pascal Plus Data Structures, Algorithms, and Advanced Programming* was always on the leading edge of computer science education and practice. With *Ada Plus Data Structures: An Object-Based Approach*, we present the next generation of software practices to the first-year student—the object paradigm.

The object paradigm is the latest step in the evolution of software development methodologies. Based on a foundation of abstraction and information hiding, the object paradigm uses a loosely coupled set of objects that interact with each other to model reality more easily. The object class extends the power of the traditional abstract data type to facilitate software design, implementation, maintenance, and reuse.

The object paradigm is still maturing. There is a good deal of debate among software engineers on the best ways to apply object technology and even more debate among educators on how to teach it. We offer an introduction to the object paradigm balanced by a presentation of traditional abstract data types, computer science theory, and concepts derived from software engineering practices. Our approach provides a strong foundation on which students can build in future courses.

An understanding of theoretical concepts helps students put new ideas that they encounter into place, and practical advice allows them to apply what they have learned. To teach these concepts to students who may not have completed college-level mathematics courses, we have used intuitive explanations, even for topics that have a basis in mathematics, such as the analysis of algorithms. In all cases, our highest goal has been to make our explanations as readable and as easily understandable as our programs.

Three Levels of Abstraction

The recurring theme throughout this book is *modeling with levels of abstraction*. Each traditional abstract data type is viewed as an object class from three different perspectives. The first perspective is from the *abstract level*. At this level we deal with the "what" questions. What is the class? What operations can it perform? What exceptions might an operation generate? At the *application level* we explore ways of modeling real-life objects in a specific context. We develop applications using a particular class. Finally, at the *implementation level* we look at specific representations of the class, its accessing operations, and its implementation in Ada. We develop different implementations and explain how to compare them.

We apply the principles of information hiding in our order of presentation. Students concentrate on the *use* of a class in the solution to a problem before implementation of the class is discussed. For example, when developing a post-fix calculator in Chapter 6, students cannot think of the stack as an array or a linked list; they have not yet seen any implementation of the stack class. Their only view of it is at the abstract level. With this order of presentation, students see that the traditional abstract data types are components used in the development of application software rather than as exercises in writing code for their professor. Ada's separation of specification and implementation allows students to code and execute solutions that use an abstract data type without any knowledge of its implementation.

The Ada Language and the Object Paradigm

Standard Pascal offers little support for today's software practices. We have selected Ada as the best language for teaching modern software development methods. Three major strengths of Ada for first-year students are its readability, stability, and reliability. Ada compiler validation ensures that all compilers support exactly the same language—a student can compile any example in this textbook on any platform with any compiler without making modifications to the code. Ada's strong typing and related features ensure that most errors are detected at compile time or through run-time constraints. We believe that the discipline enforced by the Ada compiler is the best tool for preparing students for future courses and industry where less restrictive programming languages are often used.

Ada 95 is an ISO and ANSI standard. It is the first object-oriented language with an international standard. Ada 95 provides support for objects, classes, inheritance, and runtime polymorphism. We have, however, elected to implement our designs in Ada 83. At the time we wrote our code, the Ada 9X compilers, like the language reference manual, were still evolving. Even as we write this preface in the first month of 1995, a complete Ada 95 compiler is not available. Although we refuse to present code to first-year students that might not compile on their systems, we do not ignore Ada 95. The features of Ada 95 that simplify our Ada 83 implementations are discussed in the appropriate places. It is an easy matter for an instructor to substitute Ada 95 constructs where appropriate.

We take an object-based approach (object-oriented without inheritance and run-time polymorphism) rather than an object-oriented approach for both pragmatic and pedagogical reasons. As mentioned above, at the time of this writing a complete Ada 95 compiler was not available. More important, we believe that the complexity that inheritance adds to a first-year student's learning process exceeds the reduction in design complexity made possible by it. Gaining a thorough understanding of objects, classes, and static polymorphism through an object-based approach provides a solid base for the development of larger projects where the benefits of inheritance and run-time polymorphism provided by an object-oriented approach are best appreciated.

Prerequisite Knowledge

We assume that the student reader has an elementary knowledge of the Ada programming language that includes the following topics:

1. Basic declarations (including predefined and programmer-defined scalar types, record types, and array types)
2. Input and output using package `Text_IO` and its generic packages (`Float_IO`, `Integer_IO`, and `Enumeration_IO`)

3. Basic control structures
4. Subprograms (functions and procedures)
5. How to use resources available in predefined and instructor-defined packages

The material in Appendix L is designed to allow students with previous Pascal experience to quickly master the Ada basics needed to use this book effectively.

Content and Organization

Chapters 1 through 3 present a breadth-first approach to software engineering. In Chapter 1 we discuss the basic goals of high-quality software and the basic principles of software engineering for designing and implementing programs to meet these goals. We review functional design techniques and introduce object-oriented design. Modularization, good programming style, documentation, and the separation of the design of a problem solution from its implementation are stressed throughout this chapter. We introduce packages as a method for implementing an object-oriented design. Finally, because there is more than one way to solve a problem, we discuss how competing solutions can be compared through the analysis of algorithms, using Big-O notation.

Chapter 2 addresses what we see as a critical need in software education—the ability to design and implement correct programs and to verify that they are actually correct. Topics covered in this chapter include the concept of "life-cycle" verification; designing for correctness using preconditions, postconditions, and loop invariants; the use of deskchecking and design/code walkthroughs and inspections to identify errors before testing; debugging techniques, data coverage (black box), and code coverage (clear or white box) approaches; unit testing, test plans, and structured integration testing using stubs and drivers. A case study shows how all these concepts can be applied to the development of a binary search procedure.

Chapter 3 begins our study of class design and implementation. We introduce the three perspectives of object classes: abstraction, application, and implementation. Then we apply these perspectives to some of Ada's predefined classes that students learned in their introductory programming course: scalar types, records, and arrays. We introduce data structures (collection classes) with a real-world example (a library). We illustrate how Ada packages are used for information hiding, introduce private types as a means of encapsulation, and explain why information hiding and encapsulation are important. We conclude the chapter with a case study that illustrates how the three levels of abstraction are used in the design and implementation of a Bingo game simulation.

In Chapter 4 we introduce the discrete set class. The set is first considered from its abstract perspective that we describe formally through a package specification. Then, before discussing any implementation details, we use the set specification to implement the Bingo Basket object used in the previous chapter's case study. At this point, students can design and implement software that *uses* the set class. Only then do we discuss the third perspective—the implementation of the discrete set class. Chapter 4 concludes with a case study that uses the set class in a solution to a realistic problem. We also use this case study to introduce the nature of a greedy solution.

Chapter 5 discusses the string class. We begin with the terminology and classification of strings: fixed-length, varying-length, bounded-length, and unbounded-length. We present a string class specification, implement a simple application that uses it, and develop a bounded-length implementation. We use the unbounded-length string class to motivate the need for dynamic allocation. We introduce the concept of dynamic allocation along with

the syntax for using Ada's access types. Our implementation of the unbounded-length string class demonstrates the value of pointers without having to introduce the additional complexities of linked structures.

Chapter 6 introduces the stack class. Again our order of presentation is abstract level (package specification), application level (a program to evaluate postfix expressions), and implementation level (package body). We present two implementations of the stack class: array-based and linked. The technique used to link nodes in dynamically allocated storage is described in detail and illustrated with figures. We analyze the two stack representations in terms of their number of source lines, their use of storage space, the efficiency of their operations using Big-O notation, and timed test runs. Finally, we revisit Ada's encapsulation mechanisms (introduced in Chapter 3) in the light of our stack class.

In Chapter 7 we introduce the FIFO queue. After specifying the FIFO queue class we develop an application to prepare freight train manifests. We use this application to review our object-oriented approach to software development. In this chapter we give a detailed look at the design considerations of selecting among multiple implementation choices. Two array-based implementations are discussed, as well as a linked queue representation. We analyze our FIFO queue implementations in terms of their number of source lines, their use of storage space, the efficiency of their operations using Big-O notation, and timed test runs. Finally, we examine techniques for testing the queue operations using a test driver.

Chapter 8 introduces linear lists that are ordered according to key value. In developing the abstract view of ordered lists we introduce the three operator classes: constructors, observers, and iterators. We use our list class specification in the design and implementation of an electronic address book. Our program uses retained data from its previous execution, which is stored in a binary file. We explain the use of sequential binary files for students who have never used them. The list class is implemented using both sequential (arraybased) and linked (dynamically allocated) representations. These two implementations are compared in detail, in terms of the size of their source code, their storage requirements, Big-O analysis of their operations, and timed test runs.

Chapter 9 continues the discussion of linked lists with a number of implementation variations: linked lists with dummy nodes (headers and trailers), circular linked lists, and doubly linked lists. The insertion, deletion, and list traversal algorithms are developed and implemented for each variation. We compare the different implementations in terms of the size of their source code, their storage requirements, Big-O analysis of their operations, and timed test runs. We conclude this chapter with a case study—the design and implementation of an unbounded natural number class.

In Chapter 10 we present alternative approaches for the storage of linked structures. First we discuss why we might want to store linked structures in an array rather than in dynamic memory. Then we modify the linked implementation of the ordered list developed in Chapter 8 to store the linked lists in an array rather than in dynamic memory. We compare the array-based and heap-based linked list implementations in terms of Big-O and timed test runs. Finally we show how direct files can be used to store linked lists that are retained between program runs. We explain direct files for students who have never used them and present a complete direct file-based linked list implementation.

In Chapter 11, our discussion of recursion gives students an intuitive understanding of the concept and then shows them how they can use recursion to solve programming problems. This chapter is suitable for introducing *or* reviewing recursion. Guidelines for writing recursive procedures and functions are illustrated with many examples. After demonstrating that by-hand simulation of a recursive routine can be very tedious, we introduce a

simple three-question technique for verifying the correctness of recursive procedures and functions. Because many students are wary of recursion, the introduction to this material is deliberately intuitive and nonmathematical. A more detailed discussion of how recursion works leads to an understanding of how recursion can be replaced with iteration and stacks. The case study at the end of this chapter is a recursive solution of a maze problem. We compare this implementation to a nonrecursive (stack-based) approach to demonstrate how recursion can simplify the solution to some kinds of problems.

Chapter 12 introduces binary search trees as a way to arrange data, giving the flexibility of a linked structure without the liability of slow, sequential access to its elements. We develop the tree operations in detail, then implement them using dynamic allocation with pointer (access type) variables. We present both recursive and nonrecursive versions of the insertion, deletion, retrieve, and modify operations, reinforcing the use of recursion for simplifying programming problems.

Chapter 13 presents a collection of other branching structures: heaps, priority queues (implemented with heaps), and graphs. The coverage of graphs includes the specification of a Graph class, its use in an airline application (connections between cities), and the implementation of basic graph operations with an adjacency matrix. We create both depth first and breadth-first search procedures using the Graph, Stack, and FIFO Queue classes. A procedure is developed that displays the shortest paths to all nodes from a single source using the Graph and Priority Queue classes. The chapter also describes and illustrates the use of adjacency list graph representations.

Chapter 14 presents a number of sorting algorithms and asks the question, Which is best? The sorting algorithms that are illustrated, implemented (as generic procedures) and compared include the straight selection sort, two versions of the bubble sort, insertion sort, merge sort, quick sort, and heap sort. We analyze the storage requirements of each of the sorting algorithms and efficiency in terms of both Big-O and timed test runs. Finally, to illustrate a completely different kind of sorting algorithm (one that does not compare keys), we design and implement a generic radix sort procedure.

Chapter 15 continues the discussion of algorithm analysis in the context of searching. Various searching algorithms are explained, implemented, and compared, including the sequential and binary searches developed and used earlier in the book. We introduce hashing and discuss the most common hash functions and collision resolution techniques.

Additional Features

Chapter Goals

A set of goals is presented at the beginning of each chapter to help students assess what they have learned. These goals are tested in the exercises at the end of each chapter.

Case Studies

In most chapters we develop an application program to illustrate the use of a particular class. These applications are short enough that students can follow them without being overwhelmed by a multitude of details. In addition to these application programs, we present six larger-scale case studies. Program reading is an essential skill for software professionals, but few books include programs of sufficient length for students to get this experience. The case studies provide an opportunity to follow the specification, design, and implementation of a solution to a nontrivial problem. They also provide a base for class discussion on design issues and programming assignments.

Complete Code

We include all of the Ada code for each class discussed in the text—both in print and on the program disk provided with the text. Having the source code for the abstract data types on disk encourages students to think in terms of reusable code. The source code for the application programs and case studies is provided to give students practice in modifying programs without having to spend time rekeying the original program.

Chapter Exercises

Most chapters have more than 30 exercises. The exercises have varying levels of difficulty, including short coding problems, the analysis of algorithms, and problems that test students' understanding of concepts. For chapters that contain case studies, there are exercises that specifically pertain to the material in the case study. These exercises are designed to motivate students to read the case studies carefully. Approximately one-third of the exercises are answered in the back of the text; the answer key for the remaining exercises is in the *Instructor's Guide*.

Programming Assignments

A set of programming assignments for each chapter is included at the end of each chapter. The assignments represent a range of difficulty levels and were carefully chosen to illustrate the techniques described in the text. These assignments, which include modifications and enhancements to the application programs and case studies, give students experience in program modification and program "maintenance." A further selection of programming assignments is available in the *Instructor's Guide*.

Instructor's Guide

An *Instructor's Guide* is available with suggestions for how to teach the material covered in each chapter, a key to answers to the chapter exercises not in the book, a set of questions for creating tests, and transparency masters for classroom teaching.

Acknowledgments

It is a pleasure to be able to thank the many people who helped us turn a list of good ideas into a real book. We have been extremely lucky to have many technical reviewers whose comments, corrections, and suggestions have enormously enriched this book. The successful completion of this text is largely due to the scrupulous attention of our eagle-eyed reviewers: John C. Arch, Shippensburg University; Jack Beidler, University of Scranton; Ronald L. Carlisle, Oglethorpe University; Jose Cordova, Southern Arkansas University; John Crenshaw, Western Kentucky University; Ernest Ferguson, Southwest Baptist University; Charles E. Frank, Northern Kentucky University; George C. Harrison, Norfolk State University; Charles C. Kirkpatrick, Parks College of Saint Louis University; Artur J. Kowalski, New Jersey Institute of Technology; Bill Kraynek, Florida International University; Herbert Mapes, Gallaudet University; James McCaffrey, Hawaii Pacific University; Melvin Neville, Northern Arizona University; Thomas M. Phillips, Auburn University; Paul D. Phillips, Mt. Mercy College; David A. Retterer, Ohio Northern University; Scott Sigman, Southwest Baptist University; Robert Steigerwald, US Air Force Academy; Ricky E. Sward, US Air Force Academy; Curt M. White, Indiana University-Purdue University, Fort Wayne; and Phyllis Williams, East Stroudsburg University.

We are indebted to the students enrolled in CSC 223 at the State University of New York at Plattsburgh for enduring nearly two years of working with an evolving manuscript. Their comments were invaluable in making this book accessible to future students.

The staff at D. C. Heath has been extraordinarily helpful. We particularly thank our developmental editor, Karen Jolie, for her dedication to this project and the encouragement.

she provided us while doing the job of three people. We also thank editorial assistant Heather Monahan and acquisitions editor Randall Adams. Finally, we thank our production editor, Janice Molloy, who was responsible for turning a very large stack of manuscript pages into a beautiful book.

Anyone who has ever written a textbook knows the amount of time and effort that goes into such a project. Anyone who is related to a textbook author can tell you at whose expense that time is spent. Many thanks to Naomi who, while writing her own book, cheerfully shared the computing facilities in the McCormick house.

N. D.

S. L.

J. M.

Contents

1	1
Introduction to Software Engineering	
Beyond Programming	2
A Programmer's Toolbox	3
The Goal: Quality Software	3
Goal 1: Quality Software Works	4
Goal 2: Quality Software Can Be Read and Understood	4
Goal 3: Quality Software Can Be Modified	5
Goal 4: Quality Software Is Completed on Time and within Budget	5
Specification: Understanding the Problem	6
Writing Detailed Specifications	6
Cookies for Uncle Sam	9
Program Design	10
Abstraction	10
Levels of Abstraction	11
Information Hiding	12
When Ignorance Is Bliss	12
Functional Decomposition	13
Object-Oriented Design	14
Implementation	16
The Package	16
Package Syntax	17
Packages for Classes	18
Comparing Algorithms	20
Big-O	22
Some Common Orders of Magnitude	23
Family Laundry	24

Summary	26
Exercises	27
Programming Problems	29
2	31
Verifying, Testing, and Debugging	
Where Do Bugs Come From?	32
Errors in the Specifications and Design	33
Compile-Time Errors	35
Run-Time Errors	36
Designing for Correctness	37
Assertions and Program Design	37
Preconditions and Postconditions	38
Loop Invariants	39
Deskchecking, Walk-Throughs, and Inspections	42
Program Testing	44
Developing a Testing Goal	49
Data Coverage	49
Code Coverage	50
Test Plans	52
Structured Integration Testing	52
Bottom-Up Testing	53
Top-Down Testing	55
Combining the Approaches	55
Practical Considerations	56
Debugging with a Plan	56
Summary	57
Case Study: The Binary Search and Its Test Driver	59
Searching	59
The Binary Search Algorithm	60
The Code—Version 1	62
Developing a Test Driver	63
Developing a Test Plan	66
The Code—Version 2	68
The Code—Final Version	69
Exercises	70
Programming Problems	75

3	79
Class Design and Implementation	
Three Perspectives	80
Scalar Types	80
Composite Types	82
Records	82
Abstract Level	82
Application Level	83
Implementation Level	84
Record Discriminants	85
One-Dimensional Arrays	88
Abstract Level	88
Application Level	90
Implementation Level	90
Two-Dimensional Arrays	93
Abstract Level	93
Application Level	95
Implementation Level	95
Data Structures	95
Encapsulation	99
Abstract Data Objects	102
Abstract Data Types	108
Summary	116
Case Study: Bingo Games—How Long Should They Take?	117
Exercises	123
Programming Problems	127
4	129
Sets	
The Abstract Level	130
Set Operations	131
A Set Specification	132
The Application Level	135
Bingo Number Selection	135
The Implementation Level	137
Analysis of the Set Algorithms	142
Programming for Reuse: Generic Units	142
Generic Formal Types	146

Discrete Types	146
Integer Types	149
Float Types	151
Array Types	151
Generic Formal Private Types	154
Generic Formal Subprograms	155
Case Study: A Reusable Binary Search Procedure	157
Generalizing the Index	158
Generalizing the Components	163
Private Types and Generic Formal Private Types	166
Summary	167
Case Study: Minimizing Translations	167
An Optimal Solution	168
A Greedy Solution	169
Exercises	172
Programming Problems	174
5	175
Strings	
String Terminology	176
The Abstract Level	176
A Bounded-Length String Specification	177
The Application Level	180
Alphabetizing Names	180
Creating Additional Operations	184
The Implementation Level	185
Analysis of the String Algorithms	190
Ada 83 Implementation Note	191
Access Types	193
Dynamic Memory Allocation	193
The Null Access Value	194
Using the Allocator New	194
Accessing Data through Access Values	195
Using Unchecked_Deallocation	196
Why Is It Called Unchecked_Deallocation?	197
Exceptions and Access Types	198
Access of Structured Types	198

An Unbounded-Length String Class	200
The Abstract Level	202
The Application Level	205
The Implementation Level	208
Organization of Memory	215
Ada 95 Implementation Note	215
Summary	216
Exercises	217
Programming Problems	220
6	221
Stacks	
The Abstract Level	222
What Is a Stack?	222
Operations on Stacks	222
Exceptions	225
Exceptions and Postconditions	225
The Application Level	226
Evaluating Postfix Expressions	226
Invalid Expressions	230
Other Stack Applications	234
The Implementation Level	235
The Implementation of a Stack As a Static Array	235
Array-Based Stack Package Body	236
The Implementation of a Stack As a Linked Structure	241
Implementing Procedure Push	241
Implementing Procedure Pop	244
Overflow, Underflow, Empty, and Full	247
Implementing Procedure Clear	247
Linked Stack Package Body	248
Comparing the Stack Implementations	251
Encapsulation Revisited	253
Private or Limited Private?	254
Summary	253
Exercises	255
Programming Problems	263

7	267
FIFO Queues	
The Abstract Level	268
What Is a Queue?	268
Operations on FIFO Queues	269
The Application Level	271
Freight Train Manifests	272
Object Classes	273
Algorithm Design	274
The Implementation Level	278
The Implementation of a Queue As a Static Array	278
Another Array Design	279
Comparing Array Implementations	283
The Implementation of a Queue As a Linked Structure	284
Linked Queue Package Body	287
A Circular Linked Queue Design	290
Comparing the Queue Implementations	290
Testing the Queue Operations	294
Summary	295
Exercises	295
Programming Problems	303
8	307
Linear Lists	
The Abstract Level	308
Operator Classes	308
A List Package Declaration	309
Sample Package Instantiations	312
Sample Iterator Instantiations	313
The Application Level	314
An Electronic Address Book	314
The Design	316
The Implementation	320
The Implementation Level	326
Sequential List Implementations	328
Finding a List Element	330
Using Loop Invariants	332
The Retrieve and Modify Operations	334

The Insert Operation	334
The Delete Operation	336
The Traverse Operation	338
Notes on the Sequential List Implementation	338
Sequential List Package Body	339
Linked List Implementations	342
Implementing a Linked List	343
The Traverse Operation	344
The Length Operation	344
Finding a List Element	344
The Retrieve and Modify Operations	348
The Insert Operation	348
The Delete Operation	352
Linked List Package Body	354
Analyzing the List Implementations	358
Other Factors to Consider	361
Testing the List Operations	362
Summary	362
Exercises	363
Programming Problems	370
9	375
Lists Plus	
Linked Lists with Dummy Nodes	376
The Empty Operation	379
Traversing a List with a Header and Trailer	379
Finding a List Element in a List with a Header and Trailer	380
Inserting and Deleting from a List with a Header and Trailer	385
Header, Trailer, Both, or Neither?	386
Circular Linked Lists	386
Traversing a Circular List	388
Finding a List Element in a Circular List	389
Inserting into a Circular List	392
Deleting from a Circular List	395
Doubly Linked Lists	397
Finding a List Element in a Doubly Linked List	398
Operations on a Doubly Linked List	400
Doubly Linked Lists with Dummy Nodes	403

Comparison of Implementations	405
Advanced Iterators	406
Lists with Duplicate Elements	409
Case Study: Unbounded Natural Numbers	411
The Design	413
The Implementation	414
Summary	425
Exercises	425
Programming Problems	431
10	437
Alternative Storage of Linked Structures	
The Linked List As an Array of Records	438
Why Use an Array?	438
How Is an Array Used?	438
Array Memory Management	448
Analysis of the Array-Based List	450
The Linked List As a File of Records	452
Direct Files	452
The Abstract Level	454
The Implementation Level	455
Analysis of the File-Based List	464
Summary	464
Exercises	464
Programming Problems	466
11	469
Programming with Recursion	
"Don't Ever Do This!"	470
The Classic Example of Recursion	470
Programming Recursively	473
Coding the Factorial Function	473
Verifying Recursive Procedures and Functions	475
The Three-Question Method	476
Writing Recursive Procedures and Functions	476
Writing a Boolean Function	477
Multiple Recursive Calls	479

Using Recursion to Simplify Solutions	481
Recursive Processing of Linked Lists	483
A Recursive Version of Binary Search	483
How Recursion Works	487
Static Storage Allocation	488
Dynamic Storage Allocation	490
Parameter Passing	496
Debugging Recursive Routines	497
Removing Recursion	497
Iteration	497
Stacking	499
Deciding Whether to Use a Recursive Solution	500
Summary	503
Case Study: Escape from a Maze	504
The Design	506
Escape Processing	507
The Boundary Problem	511
Finishing the Design	512
The Implementation	513
Testing the Program	517
The Recursive Solution Versus a Nonrecursive Solution	518
Exercises	519
Programming Problems	526
12 Binary Search Trees	533
The Abstract Level	534
A Binary Search Tree Package Declaration	538
Binary Tree Traversals	541
Inorder Traversals	541
Preorder Traversal	542
Postorder Traversal	543
The Application Level: Medical Emergency Response	543
The Design	545
The Implementation	548

The Implementation Level	554
Recursive Binary Search Tree Operations	555
The Size Operation	555
The Retrieve and Modify Operations	556
The Insert Operation	557
The Delete Operation	562
The Traverse Operation	568
The Clear Operation	569
Iterative Binary Search Tree Operations	570
Searching a Binary Search Tree	570
The Insert Operation	572
The Delete Operation	575
The Retrieve and Modify Operations	577
Recursion or Iteration	577
Comparing Binary Search Trees to Linear Lists	578
Big-O and Execution Time Comparisons	578
Summary	580
Exercises	581
Programming Problems	588
13	595
More Trees and Graphs	
A Nonlinked Representation of Binary Trees	596
Heaps	598
A Heap Package Specification	602
A Heap Package Body	603
Priority Queues	608
The Abstract Level	608
The Application Level: Work Request Management	610
The Implementation Level	611
Heaps Versus Other Representations of Priority Queues	614
Graphs	615
The Abstract Level	615
The Application Level: Graph Traversals	622
Depth-First Searching	622
Breadth-First Searching	626
The Single-Source Shortest Paths Problem	629
The Implementation Level	632

An Array-Based Implementation	632
Linked Implementations	638
Summary	641
Exercises	641
Programming Problems	649
14	653
Sorting Algorithms	
A Generic Sort Declaration	654
$O(N^2)$ Sorts	655
Straight Selection Sort	655
Analyzing Selection Sort	659
Bubble Sort	660
Analyzing Bubble Sort	662
Insertion Sort	665
Analysis of Insertion Sort	667
$O(N \log_2 N)$ Sorts	667
Merge Sort	668
Merging the Sorted Halves	669
Analyzing Merge Sort	672
Quick Sort	674
Splitting the Array: The Abstract Level	674
Splitting the Array: The Implementation Level	676
Analyzing Quick Sort	679
Heap Sort	680
Building a Heap	680
Sorting the Heap	682
Analyzing Heap Sort	684
Efficiency Considerations	685
When N Is Small	685
Eliminating Calls to Procedures and Functions	686
In-Line Subprograms	686
Removal of Recursion	687
More about Sorting in General	687
Sorting in Descending Order	687
Keys	688
Sorting Pointers	689

Radix Sort	691
The Implementation Level	694
Analysis of Radix Sort	696
Summary	697
Exercises	698
Programming Problems	704
15 Searching	707
Linear Searching	708
High-Probability Ordering	710
Key Ordering	710
Binary Searching	712
Hashing	713
Collisions	716
Linear Probing	717
Clustering	720
Rehashing	720
Buckets and Chaining	722
Choosing a Good Hash Function	724
Division Method	725
Other Hash Methods	726
A Hash-Based List Package	730
Summary	730
Exercises	731
Programming Problems	735
Appendix A Ada Reserved Words	739
Appendix B Ada Syntax (EBNF)	740
Appendix C ASCII Character Set	752
Appendix D Specification of Package Text_IO	754
Appendix E Specifications of Packages Sequential_IO and Direct_IO	759
Appendix F Specification of the Package Calendar	762
Appendix G Specification of the Package Standard	763
Appendix H Program Style, Formatting and Documentation	768

1—

Introduction to Software Engineering

GOALS

After reading this chapter, you should be able to

- describe the general activities in the software life cycle and the goals for "quality" software
- explain the terms software requirements, software specifications, software design, information hiding, abstraction, and encapsulation
- understand the need for different levels of abstraction
- understand the fundamental ideas of functional decomposition and object-oriented design
- read simple Ada packages

Beyond Programming

When we talk about computer programming, we immediately think of writing a program for a computer to execute—the generation of code in some computer language. As a beginning student of computer science, you wrote programs that solved relatively simple problems. Much of your effort at the start went into learning the syntax of a programming language such as Ada: the language's reserved words, its data types, its constructs for selection (`if then else` and `case`) and looping (`loop`, `while`, and `for`), and its input/output mechanisms (`Text_IO`, `Integer_IO`, `Float_IO`, `Sequential_IO`, and `Direct_IO`).

You may have learned a programming methodology that took you from the problem description your instructor handed out all the way through to the delivery of a good software solution. There are many design techniques, coding standards, and testing methods that programmers before you have come up with to help develop high-quality software. But why bother with all that methodology? Why not just sit down at a computer and write programs? Aren't we wasting a lot of time and effort on technique, when we could just write our programs directly in Ada, C, Pascal, or Smalltalk?

If the degree of our programming sophistication never had to rise above the level of trivial programs (like summing a list of prices or averaging grades), we might get away with such a code-first technique (or rather, *lack* of technique). Some new owners of personal computers program this way, hacking away at the code until the program works more or less correctly.

As your programs grow larger and more complex, however, you must pay attention to other software issues in addition to coding. If you become a software professional, someday you may work as part of a team that develops a system containing tens of thousands, or even millions, of lines of code. The activities involved in such a software project's whole "life cycle" clearly go beyond just sitting down at a computer and writing programs. These activities include

- *Problem analysis* understanding the nature of the problem to be solved
- *Requirements definition* specifying exactly what the program must do
- *High- and low-level design* recording how the program meets the requirements, from the "big picture" overview to the detailed design
- *Implementation of the design* coding a program in a computer language
- *Testing and verification* detecting and fixing errors and demonstrating the correctness of the program
- *Delivery* turning over the tested program to the customer or user (or instructor!)
- *Operation* actually using the program
- *Maintenance* making changes to fix operational errors and to add or modify functions of the program

Software development is not simply a matter of going through these steps one after another. Many of these activities take place concurrently. We may be coding one part of the solution while we're designing another part, or defining requirements for a new version of a program while we're still testing the current version. Often a number of people work on different parts of the same program simultaneously. Keeping track of all these activities is not an easy task.

Software Engineering

A disciplined approach to the design, production, and maintenance of computer programs that are developed on time and within cost estimates, utilizing tools that help to manage the size and complexity of the resulting software products.

The term **software engineering** is used to refer to a disciplined approach to the development of computer programs, through *all* of these software life-cycle activities. What makes our jobs as programmers or software engineers challenging is the tendency of software to grow in size and complexity and to change at every stage of its development. Part of the discipline in the software engineering approach is the utilization of tools to manage this size and complexity to develop high-quality software.

A Programmer's Toolbox

What tools does a programmer use? Usually a programmer has several toolboxes, each containing tools that help to build and shape a software product.

Hardware

One toolbox contains the hardware itself: the computers and their peripheral devices (such as monitors, terminals, storage devices, and printers), on which and for which we develop software.

Software

A second toolbox contains various software tools: operating systems to control the computer's resources, text editors to help us enter programs, compilers to translate high-level languages like Ada into something the computer can execute, interactive debugging programs, test data generators, and so on. You've used some of these tools already.

Ideaware

A third toolbox is filled with the shared body of knowledge that programmers have collected over time. This box contains the algorithms that we use to solve common programming problems, as well as data structures for modeling the information processed by our programs. It contains programming methodologies such as object-oriented design and software concepts including information hiding, data encapsulation, and abstraction. There are also some tools for measuring, evaluating, and proving the correctness of our programs. We spend most of the rest of this book exploring the contents of this third toolbox.

It might be argued that the use of these tools takes the creativity out of programming, making the design of software a cut-and-dried activity. We don't believe that to be true. Artists and composers are creative, yet their innovations are grounded in the basic principles of their crafts. Similarly, the most creative programmers build high-quality software through the disciplined use of basic programming tools.

***The Goal:
Quality Software***

Writing a program that does something is not enough. A programmer must determine what the software is supposed to do and then write a good program that accomplishes the task. But what constitutes a good program?

Quality software achieves these goals:

1. It works.
2. It can be read and understood.
3. It can be modified, if necessary, without excruciating time and effort.
4. It is completed on time and within budget.

Meeting these goals isn't easy, but they are all important. Let's talk a little about each one.

Goal 1— Quality Software Works

The program must do the task it was supposed to perform, correctly and completely. The first step in the development process is to determine exactly what the program is required to do. If you don't know that, you cannot write a program that works.

Requirements

A statement of what is to be provided by a computer system or software product.

In order to know what the program is supposed to accomplish, we must have a definition of the program's **requirements**. For students, the requirements often come from the instructor's problem description: "Write a program that calculates. . . ." For programmers on a government contract, the requirements document may be hundreds of pages long.

Software Specifications

Detailed description of the function, inputs, processing, outputs, and special requirements of a software product, which provides the information needed to design and implement the program.

We develop programs that meet the user's requirements by using **software specifications**. The *specifications* indicate the format of the input and the expected output, details about processing, performance measures (how fast? how big? how accurate?), what to do in case of errors, and so on. The specifications tell exactly *what* the program does, but not *how* it is done. Sometimes your instructor gives you detailed specifications; other times you have to write them yourself, based on the requirements definition, conversations with your instructor, or guesswork. We discuss this issue in more detail later in the chapter.

How do you know when the program is right? A program should accurately satisfy every item in its requirements—so the program must be *complete* (it should "do everything") and *correct* (it should "do it right").

Goal 1 also means that the program should be *usable*. If the program needs data from a person sitting at a terminal, it must tell the user when it expects input. The program's outputs should be readable and understandable; they should make sense to human beings. Creating a good user interface is an important subject in software engineering today.

Finally, Goal 1 means that the program should be as *efficient as it needs to be*. We never go out of our way to write programs that waste time or space in memory, but not all programs demand a specific level of efficiency. When they do, however, we must meet these demands or else the programs do not satisfy the problems' requirements. For instance, a space-launch control program must execute in "real time"; that is, the software must process commands, perform calculations, and display results in pace with the activities it is supposed to control. Closer to home, if a desktop publishing program cannot update the screen as fast as the user can type, the program is not as efficient as it needs to be. If the software isn't efficient enough, it doesn't meet its requirements, and thus, according to our definition, it doesn't work correctly.

Goal 2— Quality Software Can Be Read and Understood

The first goal says that the computer executes the program as expected; the second goal is more concerned with the human reading it.

You might wonder why anyone would want to read a computer program. Chances are you have already read a number of them in programming textbooks. Reading a well-written program can teach you techniques that help you write good programs. In fact, it's hard to imagine how a person could become a good programmer *without* reading good programs.

Think of the last time you had to debug one of your programs; you probably spent quite a bit of time reading it. A well-designed and clearly written program is certainly easier for human readers to understand. Later in this chapter we describe a software design methodology that produces "structured," well-documented programs that people, as well as computers, can understand. The number of pages of documentation required for "real world" programs usually exceeds the number of pages of Ada code. Most every organization has its own policy for documentation. Appendix H contains our suggestions for writing programs that are easy to read.

Although the computer is the ultimate reader of the program, it is likely that a program that cannot be understood by its human authors contains errors.

Goal 3—

Quality Software Can Be Modified

When does software need to be modified? Changes occur in every phase of its existence.

Software gets changed in the design phase. When your instructor or employer gives you a programming assignment, you begin to think of how to solve the problem. The next time you meet, however, you may be notified of a small change in the program description.

Software gets changed in the coding phase. You make changes in your program as a result of compilation errors. Sometimes you suddenly see a better solution to a part of the problem after the program has been coded, so you make changes.

Software gets changed in the testing phase. If the program crashes or gives incorrect results, you must make corrections.

In an academic environment, the life of a piece of software typically ends when the corrected program gets turned in to be graded. When software is being developed for use, however, most of the changes that are made take place during the maintenance phase. Someone may discover an error that wasn't uncovered in testing, someone else may want additional functions, a third party may want to change the input format, and a fourth may want to run the program on another system.

The point is that software changes often and at all phases of its life cycle. Knowing this, software engineers try to develop programs that are easily modifiable. If you think it's easy to change a program, try to make a small change in the last program you wrote. It's difficult to remember all the details of a program after some time has passed. Modifications to programs often are not even made by the original authors but by subsequent maintenance programmers. (Someday you may be the one making the modifications to someone else's program.)

What makes a program easily modifiable? First, it should meet Goal 2: it should be readable and understandable to humans. Second, it should be able to survive little changes without becoming a big mess. The design methodology we introduce later in this chapter should help you write programs that meet these goals.

Goal 4—

Quality Software is Completed on Time and within Budget

You know what happens in school when you turn your program in late. You probably have grieved over an otherwise perfect program that was given only half credit because you turned it in one day late. "But the network was down five hours last night!" you protest.

Although the consequences of tardiness may seem arbitrary in the academic world, they are significant in the business world. The software for controlling a space launch must be developed and tested before the launch can take place. A patient database system for a new hospital must be installed before the hospital can open. In such cases the program doesn't meet its requirements if it isn't ready when needed.

"Time is money" may sound trite but failure to meet deadlines is *expensive*. A company generally budgets a certain amount of time and money for the development of a piece of software. As a programmer, you are paid a salary or hourly wage. If your part of the project is only 80% complete when the deadline arrives, the company must pay you—or another programmer—to finish the work. However, the extra expenditure in salary is not the only cost. Other workers may be waiting to integrate your part of the program into the system for testing. If the program is part of a contract with a customer, there may be monetary penalties for missed deadlines. If the program is being developed for

commercial sales, your company may lose money if another firm puts a similar product on the market first.

Once you know what your goals are, what can you do to meet them? Where should you start? There are many tools and techniques used by software engineers. In the next few sections of this chapter, we focus on techniques to help you understand, design, and code programs.

Specification:

Understanding the Problem

No matter what programming design technique you use, the first steps are the same. Imagine the following all-too-familiar situation. On the third day of class, you are given a 12-page description of Programming Assignment 1, which must be running perfectly and turned in by noon, a week from yesterday. You read the assignment and realize that this program is three times larger than any program you have ever written. Now, what is your first step?

The responses below are typical of those given by a class of computer science students in such a situation:

1. Panic	39%
2. Sit down at the computer and begin typing	26%
3. Drop the course	24%
4. Copy the code from a smart classmate	7%
5. Stop and think	4%

Response 1 is a reasonable reaction from students who have not learned good programming techniques. Students who answer with Response 3 find their education coming along rather slowly. Response 4 usually gets you scholastic probation at most universities. Response 2 seems like a good idea, considering the deadline looming ahead. But resist the temptation to reach for a keyboard; the first step is to *think*. Before you can come up with a program solution, you must understand the problem. Read the assignment, and then read it again. Ask questions of your instructor (or manager, or client).

The problem with writing first is that it tends to lock you into the first solution you think of, which may not be the best approach. There is a natural tendency to believe that once you've put something in writing, you've invested too much in the idea to toss it out and start over.

On the other hand, don't simply think over all the possibilities until the day before your deadline. (Chances are a disk drive, network, or printer will fail that day!) When you think you understand the problem you should begin writing.

Writing Detailed Specifications

Many writers feel a moment of terror when faced with a blank piece of paper—where to begin? As a programmer, however, you don't have to wonder about where to begin. You can make the first mark and get on with the job by doing an important preliminary task.

Using the assignment description (your "requirements"), write a complete definition of the problem; include the details of the expected inputs and outputs, the necessary processing and error handling, and all the assumptions about the problem. When you finish this task, you have a *detailed specification*—a formal definition of the problem your pro-

gram must solve, telling you exactly what the program should do. In addition, the process of writing the specifications brings to light any holes in the requirements. For instance, are embedded blanks in the input significant or can they be ignored? Do you need to check for errors in the input? On what computer system(s) will your program be run? If you get the answers to these questions at this stage, you can design and code your program correctly from the start.

Many software engineers make use of user/operational *scenarios* to understand the requirements. In software design, a scenario is a sequence of events for *one* execution of the program. Here, for example, is a scenario that a designer might consider when developing the software for a bank's automated teller machine (ATM).

1. The customer inserts bank card.
2. The ATM reads the account number on the card.
3. The ATM requests a PIN (personal identification number) from the customer.
4. The customer enters 5683.
5. The ATM successfully verifies the account number PIN combination.
6. The ATM asks the customer to select a transaction type (deposit, show balance, withdrawal, or quit).
7. The customer selects show balance.
8. The ATM obtains the current account balance (\$1,204.35) and displays it.
9. The ATM asks the customer to select a transaction type (deposit, show balance, withdrawal, or quit).
10. The customer selects quit.
11. The ATM returns the customer's bank card.

Scenarios allow us to get a feel for the behavior expected from the system. A single scenario cannot show all possible behaviors of the system. Software engineers typically prepare many different scenarios in order to gain a full understanding of the system's requirements.

Sometimes, but not always, your instructor gives you an assignment description that also can function as a detailed specification. Other times you receive "simple" program specifications, and the pertinent details must be divined by ESP or by hanging around the professor's office. Consider the following directions:

Write a program to count the lines of source code in an Ada program. The input is an Ada program. The output shall be a listing of the program, with line numbers preceding each executable line of source code, followed by a report of the line counts.

The programming task described is not complicated, but some important information is missing from these requirements. When you write a detailed specification, you must answer these questions:

Input: Where is the program to be processed? In a single text file? Does the user provide the name of the file?

Output: Where? To a file or to the screen? What goes in the "report of the line counts"?

Processing requirements: Does it matter whether the program to be processed is syntactically correct? What is meant by "line numbers preceding each executable line of source code"? Does that mean that blank lines and comment lines should not be

counted? Is a declaration "executable"? Will this program be compiled and run on a certain computer system (for example, IBM PC-compatible computers) or on different systems (both personal computers and mainframes)?

You must know some details in order to write and run the program. Other details, if not explicitly stated in the program's requirements, may be handled according to the programmer's preference. Assumptions about unstated or ambiguous specifications always should be written explicitly in the program's documentation.

The detailed specification clarifies the problem to be solved. But it does more than that: It also serves as an important piece of written documentation about the program. A detailed specification for the Program `Listener` is shown in Figure 1-1. There are many ways specifications may be expressed and a number of different sections that may be included, depending on the nature of the problem. Our recommended program specification includes the following sections: Inputs, Outputs, Processing Requirements, and Assumptions. If special processing is needed for unusual or error conditions, it too should be specified. Sometimes it is helpful to include a section containing definitions of terms used in the specification. It is also useful to list any testing requirements so that the method for verifying the program is taken into consideration early in the development process.

The need for accurate, readable specifications is not limited to software engineering. Sometimes even a simple cookie requires a detailed specification (see Cookies for Uncle Sam on page 9).

Specification: Program `Listener`

Function

This program shall process the lines of source code in an Ada program, producing a listing of the program, with line numbers preceding each executable line of source code, followed by a report of the counts of executable and comment source lines of code.

Definitions

Line of source code: A line of an Ada program file (all characters up to the line terminator), which may contain statements or parts of statements, comments, and/or blanks.

Blank line: A line of source code that does not contain any nonblank characters.

Executable line: A line of source code that contains Ada declarations, statements, or reserved words. (Note: It may contain comments in addition to functional code.)

Comment line: A line of source code that contains *only* an Ada comment.

Input

The Ada program to be processed is contained in a single text file. The user should be prompted to supply the file specification.

Output

The lines shall be displayed on the screen as they are read from the source file. The formatted report output shall be written to a file called COUNTOUT. The output shall contain two sections:

1. The *Program Listing* shall consist of all the lines of the Ada program. Three types of lines are treated as follows:

- (a) Blank line: Print a blank line.
- (b) Executable line: The line shall be printed, preceded by the current line count. The line count value shall be printed in the first four columns and shall be separated from the Ada code by four blank spaces.
- (c) Comment line: The line shall be printed preceded by eight blank spaces (to maintain program formatting and indentation).

2. The Count Report shall consist of the following elements, printed with appropriate labels:

- (a) Number of executable lines of code
- (b) Number of comment lines

Processing Requirements

1. This program must be able to differentiate between executable and comment lines. Executable and comment lines shall be counted separately; only executable lines shall be preceded by a line number in the program listing.
2. This program must be able to be compiled and run on any computer system with an Ada compiler.

Assumptions

1. The Ada program in the input file is syntactically correct.
2. The program contains only printable characters, including blanks. (It may not contain tabs.)
3. A program line contains no more than 124 characters.
4. A program contains no more than 9999 executable statements.

Figure 1-1
Specification for the Source Lister and Counter

Cookies for Uncle Sam

What sort of commercially made cookies are suitable for the tastes of American armed forces personnel? Here are excerpts from the 15-page specification issued by the U.S. military procurement offices:

Type I, oatmeal with chocolate chips. The cookies shall be well-baked. They shall be browned on the bottom surface and outer edges, but not appreciably browned on the top surface. They shall be wholly intact, free of checks or cracks. . . . The cookies shall be tender and crisp with an appetizing flavor, free of a burnt or scorched flavor. . . .

Type II, sandwich. Each cookie shall consist of two round base cakes with a layer of filling between them. The weight of a cookie shall not be less than 21.5 grams with the filling weighing not less than 6.4 grams. The base cakes shall have been uniformly well-baked with a color ranging from not lighter than chip 27885 or darker than chip 11711. . . . The color comparisons shall

be made under . . . sky daylight with the objects held in such a way as to avoid specular refractance [glossiness]. . . . The filling shall be centered so that it does not protrude beyond the perimeter of the base edges. . . .

Program Design

The detailed specification of the program tells *what* the program must do, but not *how* it does it. Once you have fully clarified the goals of the program, you can begin to develop and record a strategy for meeting them: the design phase of the software life cycle.

Abstraction

The universe is filled with complex systems. We learn about such systems through *models*. A model may be mathematical, like equations describing the motion of satellites around the earth. A physical object such as a model airplane used in wind tunnel tests is another form of model. In this approach to understanding complex systems the important concept is that only the essential characteristics of the system are considered; minor or irrelevant details are ignored. For example, although the earth is an oblate ellipsoid, globes (models of the earth) are spheres. The small difference between the earth's equatorial diameter and polar diameter is not important to us in studying the political divisions and physical landmarks on the earth. Similarly, in-flight movies are not included in the model airplanes used for wind tunnel testing.

Abstraction

A model of a complex system that includes only the essential details relative to the perspective of the viewer of the system.

An **abstraction** is a model of a complex system that includes only the essential details. Abstractions are the fundamental way that we manage complexity. Different abstractions of a particular system are used by different viewers of the system. Thus, while we see a car as a means to transport us and our friends, the automotive brake engineer may see it as a large mass with a small contact area between it and the road (Figure 1-2).

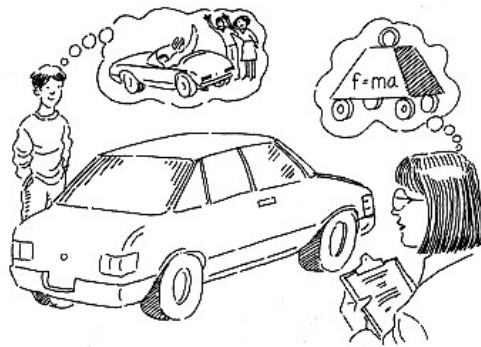


Figure 1-2
An Abstraction Includes the Essential Details Relative to the
Perspective of the Viewer

Levels of Abstraction

A single abstraction is inadequate for a *complete* understanding of a system; a set of different abstractions is needed. To manage groups of abstractions we organize them into hierarchies. The most commonly used form of hierarchy is the "part of" hierarchy. The basic idea is to break a system down into subsystems. Figure 1-3 illustrates such a hierarchy. This figure shows that a desktop computer is composed of a keyboard, a mouse, a system box, a modem, and a monitor. Alternatively, we can say that a keyboard is *part of* a desktop computer. The CPU is *part of* the system box and the control unit is *part of* the CPU. A person wanting to use a desktop computer for word processing needs to understand fewer of the levels in this hierarchy than a person wanting to assemble or service the computer.

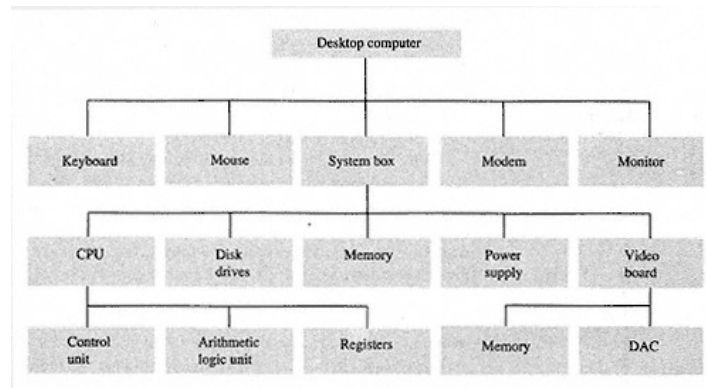


Figure 1-3
"Part of" Hierarchy

Another useful hierarchy of abstractions is the "kind of" hierarchy. We use this hierarchy to classify things from the most general to the most specific. Figure 1-4 illustrates a "kind of" hierarchy. From this hierarchy, we can see that sculpture, paintings, and prints are all *kinds of* artwork. A photograph is a *kind of* print, and a black and white photo is a *kind of* photograph.

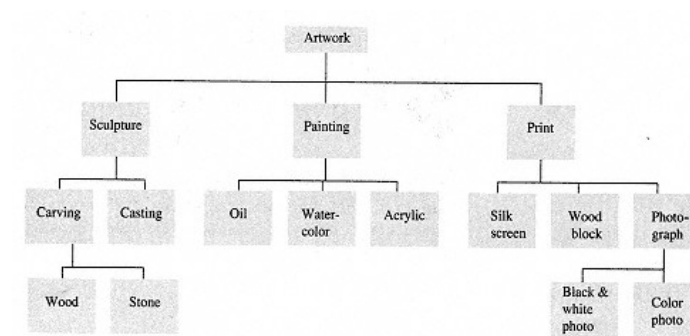


Figure 1-4
"Kind of" Hierarchy

What does abstraction have to do with software development? The programs we write are abstractions. The spreadsheet program used by an accountant models the books used to record debits and credits. The autopilot program used by a pilot models the flight characteristics of the aircraft. Writing software is difficult because both the systems we model and the processes we use to develop the software are complex. Abstraction is our most powerful tool for dealing with this complexity. One of our major goals is to show you how abstractions are used to manage the complexity of developing software. In nearly every chapter we make use of abstraction to simplify our work.

Information Hiding

Information Hiding

The practice of controlling access to the details of a module or structure.

In the next sections we briefly look at two program design methods. Both are based on decomposing the problem into clear-cut levels of abstraction. One important feature of any design method is that the details that are specified in lower levels of the program design are hidden from the higher levels. The programmer only sees the details that are relevant at a particular level of the design. This **information hiding** makes certain details inaccessible to the programmer at higher levels.

Why is the hiding of details desirable? Shouldn't the programmer know everything? *No!* This situation is one in which a certain amount of ignorance is advantageous. Information hiding prevents the high levels of the design from becoming dependent on lowlevel design details that are more likely to be changed. For example, you can stop a car without knowing whether it has disk brakes or drum brakes. You don't need to know these lower-level details of the car's brake subsystem to stop it.

Furthermore, you don't want to require a complete understanding of the complicated details of low-level routines for the design of higher level routines. Such a requirement would introduce a greater risk of confusion and errors throughout the whole program. For example, it would be disastrous if every time we wanted to stop our car, we had to think, "The brake pedal is a lever with a mechanical advantage of 10.6 coupled to a hydraulic system with a mechanical advantage of 7.3 that presses a semimetallic pad against a steel disk. The coefficient of friction of the pad/disk contact is . . ."

Information hiding is not limited to driving cars and programming computers. Try to list *all* the operations and information required to make a peanut butter and jelly sandwich. We normally don't consider the details of planting, growing, and harvesting peanuts, grapes, and wheat as part of making a sandwich. Information hiding lets us deal only with the operations and information needed at a particular level in the solution of a problem.

The concepts of abstraction and information hiding are fundamental principles of software engineering. We come back to them again and again throughout this book.

When Ignorance Is Bliss

As another illustration of the value of information hiding, consider Ada's Get procedure for floating point numbers. We use it all the time to input data, but do you know how it works? Its implementation is completely invisible to the user of the procedure. When the program says

```
Float_IO.Get (File => Data_File, Item => Real_Number);
```

this procedure somehow gets the characters that represent a floating point number from the appropriate place in `Data_File`, updates the reading marker to pass the characters that were read, converts these characters into the representation of a floating point number for this particular computer, and stores that value in the actual parameter, `Real_Number`.

Even this simplified explanation of the procedure is inconsequential to the programmer who calls `Float_IO.Get`. What is important to the programmer is to have a simple way to call the procedure and to know that `Get` assigns an appropriate value to the specified variable. The calling program doesn't have to know anything about the internal representation of the file variable or the implementation of floating-point numbers on this machine. It only needs to know the interface to procedure `Get`. If a change ever is made in the details of how this predefined procedure does what it does, we do not have to change any of our programs that call `Get`.

The software design approach we use must help us divide a complex problem into smaller, more manageable modules that are easier to build and understand. These modules should be good abstractions with strong *cohesion*; that is, each module should have a single purpose or identity and should stick together well. A cohesive module can usually be described by a simple sentence. If you have to use several sentences or a run-on sentence with lots of "ands" and "ors" to describe your module, it is probably *not* cohesive. Each module should also exhibit information hiding so that changes within it do not result in changes in modules that use it. This independent quality of modules is known as *loose coupling*. If your module depends on many other modules it is *not* loosely coupled.

But what should these modules be? One approach is to break the problem into *functional* subproblems (Do this, then do this, then do that). Another approach is to divide the problem into the "things" or objects that interact to solve the problem. We examine both these approaches.

Functional Decomposition

One method for designing software is called *functional decomposition*, also called *top-down design*. You probably learned this method in your introductory class. First the problem is broken into several large tasks. Each of these tasks is in turn divided into sections, then the sections are subdivided, and so on. The important feature is that *details are deferred as long as possible* as we go from a general to a specific solution.

The development of a computer program by functional decomposition begins with a "big picture" solution to the problem defined in the specification. We then devise a general strategy for solving the problem by dividing it into manageable functional modules. Next, each of the large functional modules is subdivided into several tasks. We do not need to write the top level of the functional design in source code (such as Ada); we can write it in English or "pseudocode." (Some software development projects even use special design languages that can be compiled.) This divide-and-conquer activity continues until we get down to a level that can be easily translated into lines of code.

Once it has been divided into modules, the problem is simpler to code into a well-structured program. The functional decomposition approach encourages programming in

logical units, using procedures and functions. The main module of the design becomes the main program, and subsections develop into procedures or functions.

As an example, let's start the functional design for making a cake.

Make Cake

Get Ingredients
 Mix Cake Ingredients
 Bake
 Cool
 Apply Icing

The problem now is divided into five logical units, each of which might be further decomposed into more detailed functional modules. Figure 1-5 illustrates the hierarchy of such a functional decomposition.

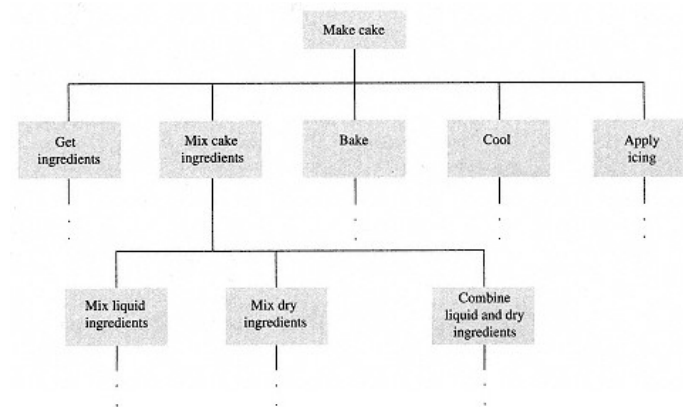


Figure 1-5
 A Portion of a Functional Design for Baking a Cake

In functional decomposition, information hiding is accomplished primarily through deferring the details of algorithms. This form of information hiding is based on *procedural* or *functional abstraction*.

Object-Oriented Design

While functional decomposition is suitable for developing solutions to problems small enough to be handled by a single programmer or a small group of programmers, it is generally inadequate for large-scale problems. We believe that the solutions to large-scale problems are better designed with another method: object-oriented design.

Object-oriented design, like functional decomposition, takes a divide-and-conquer approach. However, instead of decomposing the problem into functional modules, we divide it into entities or things that make sense in the context of the problem being solved. These entities, called *objects*, collaborate and interact to solve the problem.

Let's list some of the objects in our baking problem. There are, of course, all of the various ingredients: eggs, milk, flour, butter, and so on. We also need certain pieces of equipment such as pans, bowls, measuring spoons, and an oven. The baker is another important entity in this problem. All of these entities must collaborate to create a cake. For example, a spoon measures individual ingredients and a bowl holds a mixture of ingredients.

Object Class (Class)
 The description of a group of objects with similar properties and behaviors. A template for creating individual objects.

Groups of objects with similar properties and behaviors are described by an **object class** (usually shortened to *class*). Each oven in the world is a unique object. We cannot hope to described every oven. But we can group oven objects together into a class called "oven" that has certain properties and behaviors.

An object class is similar to an Ada type. Classes are templates for objects; Ada types are templates for variables. Like types, object classes have attributes and operations associated with them. For example, an oven class might have an attribute to specify whether it is gas or electric and operations to turn it on or off and to set it to maintain a desired temperature.

With object-oriented design, we determine the classes from the things in the problem as described in the specification. We also describe how those classes are related to each other, and how they interact with each other. From this work, we determine a set of properties (attributes) and a set of responsibilities (operations) to associate with each class. With object-oriented design, the *function* of the program is distributed among a set of collaborating objects. The following table illustrates some of the object classes that participate in baking a cake.

<i>Class</i>	<i>Attributes</i>	<i>Responsibilities (Operations)</i>
Oven	Energy Source Size Temperature Number Racks	Turn On Turn Off Set Desired Temperature
Bowl	Capacity Current Amount	Add to Dump
Egg	Size	Crack Separate (white from yolk)

Once we have defined an oven class, it can be reused in other cooking problems such as roasting a turkey. Reuse of classes is an important aspect of modern software development. One of the major purposes of this text is to introduce you to a number of classes that are particularly important in the development of software—perhaps the most reusable classes known. These classes often are called *abstract data types*. We discuss the concept of the abstract data type in detail in Chapter 3. In the remaining chapters, we fully develop many abstract data types. We described others and let you develop them. As these classes are fundamental to computer science, the Ada code for them can often be obtained from a public or private repository or purchased from vendors who market Ada components. If they are already available, why do we spend so much time on their development? Our goal is to teach you how to develop software. As with the development of any skill, you need to practice the fundamentals before you can become a virtuoso.

There is more to object-oriented design than we have given here. We discuss additional details of the design process throughout the remainder of this book. However, this text only serves as a brief introduction to object-oriented design. You will learn more in future classes and through your experience designing and implementing software.

Object-oriented design does not replace functional design methods, it is used in combination with them. We still use functional decomposition to develop the algorithms of operations performed by objects. However, we do not use functional decomposition to break down the problem into modules.

Implementation

How do we translate an object-oriented design into an Ada program? The simplest way is to use an Ada type for each class and a variable or named constant to represent each object. For example, we can identify the objects Program File and Report File in our program listing problem described in Figure 1-1. Both of these objects are members of a class we might call File. Ada's predefined type `File_Type` in package `Text_IO` can be used to implement this class. The individual objects may then be declared as variables of this type like this:

```
Program_File : Text_IO.File_Type;
Report_File  : Text_IO.File_Type;
```

Ada has a number of predefined types like `File_Type`, `String`, `Float`, and `Integer` that can be used to implement classes. However, it is not possible for any programming language to supply all of the classes we will ever need. Instead, modern high-level languages supply us with the means to construct our own classes. In Ada, we use the package to define new classes.

The Package

Encapsulation

A programming language feature that allows a compiler to enforce information hiding.

Encapsulation is a programming language feature that lets a compiler enforce information hiding. It is like the case around a watch that prevents us from accessing the works. The case is provided by the watchmaker who can easily open it when repairs become necessary. Encapsulation lets us use code reliably and with less worry. Most programming languages provide the subprogram and hierarchical record for encapsulation. Although these methods of isolating modules and data have been adequate for the programs you have written to date, they are insufficient for larger programs, particularly for programs that are complex enough to require a team of programmers. Ada was designed for developing large programs. Packages, both predefined and programmer written, are Ada's principal method of encapsulation for complex programs. With packages, we can concentrate our efforts on one level of the problem while our fellow team members concentrate on other levels. The programming process becomes more like making a sandwich. One person concentrates on spreading the bread with peanut butter and jelly; someone else is responsible for growing the peanuts.

Package

A group of logically related entities that may include types and subtypes, constants of those types and subtypes, and subprograms with parameters of those types and subtypes.

A **package** is a group of logically related entities that may include types and subtypes, constants of those types and subtypes, and subprograms with parameters of those types and subtypes. Once a package has been written and thoroughly tested, we can use it with

confidence in later programs simply by including an appropriate `with` clause. By now you are comfortable using packages. Your first Ada program probably included the statement `with Text_IO;` to declare your intent to use resources provided by the package `Text_IO`.

Packages are written in two parts, the package specification and the package body. The package specification defines the interface to the package. It is like the face and knobs on a watch. The specification describes what *resources* the package can supply to the program. Resources supplied by a watch might include the value of the current time and operations to set the current time. In a package, resources can include types, subtypes, named constants, and subprograms. The package body provides the implementation of the resources defined in the package specification; the insides of the watch.

There are significant advantages to separating the specification of a package from its implementation. A clear interface is important, particularly when a package is used by other members of a programming team. Any ambiguity in an interface results in problems when the team members' efforts are combined. By separating the specification of the package from its implementation, we are given the opportunity to concentrate our efforts on the design of an object class without needing to worry about implementation details.

By using separate specifications and interfaces for our packages we can save time during the development of our programs. When an error is found we need only correct and recompile the package body that contained the error. We do not have to recompile the entire program, a time-consuming process when our programs are large. (The issues of separate compilation are discussed in Appendix J.)

Another advantage of this separation is that we can change the implementation (the body) at any time without affecting the work done by other programmers. We can make changes when a better algorithm is discovered or there is a change in the environment in which the program is run. For example, suppose we need to control how text is displayed on a screen. Text control operations may include moving the cursor to a particular location and setting text characteristics usually differ from one computer system to another. By defining an interface and encapsulating the algorithms in the package body, we can easily move our program to a different system simply by rewriting one package body. We do not have to change the rest of the program.

Package Syntax

Here is a simplified EBNF definition of the package specification. If you are not familiar with reading EBNF definitions, read the introduction to EBNF in Appendix B.

```
package_declaration ::= package_specification;

package_specification ::= package identifier is
                        {basic_declarative_item}

                        end [package_simple_name]

basic_declarative_item ::= type_declaration           |
                        subtype_declaration          |
                        named_constant_declaration    |
                        subprogram_declaration        |
                        generic_instantiation
```

Notice that the package specification can contain subprogram declarations but not subprogram bodies. Because the package specification serves as the interface to the package, all of the declarations that it contains are available to the rest of the program. If the package specification contained the subprogram bodies, the implementations of the subprograms would not be hidden from the program.

Here is a simplified EBNF definition of the package body:

```
package_body      ::=  package body
                        package_simple_name is
                        [declarative_part]

                        end [package_simple_name];

declarative_part  ::=  {basic_declarative_item}
                        {later_declarative_item}

later_declarative_item ::=  generic_instantiation      |
                            subprogram_declaration     |
                            subprogram_body
```

The package body contains the implementation details for items declared in the package specification. It also can contain additional subtypes, types, named constants, generic instantiations (like `Integer_IO`), and subprograms that we need for these implementations. Anything *declared* in the package body is not available to the rest of the program.

In order to access the resources of `Text_`, `Direct_IO`, `Sequential_IO`, or `Calendar` from our program we include a `with` clause that named the package. This same mechanism is used to access a programmer-written package.

Packages for Classes

There are different ways to implement a class with an Ada package. In this section we present the simplest method. In later chapters we show you implementations with superior methods of encapsulation. For now, in a package specification used to implement a class we include:

1. a type declaration for the class
2. a subprogram declaration for each attribute of the class and each operation performed by objects of the class
3. any type or subtype declarations for simple types needed for parameters in the subprogram declarations

Let's develop a package specification for one of the classes in our baking problem: the oven class. This is not a farfetched example; some ovens are now controlled by micro-processors. The attributes and operations for our oven class are listed in the table on page 15. The package given in Specification 1-1 describes all of the requirements of our oven class.

Specification 1-1— The Oven Class

```
package Oven_Class is

    -- Simple types needed for oven class subprograms
    type Energy_Type is (Electric, Gas);
    subtype Cubic_Inches is Positive;
    subtype Degrees is Integer range 100..500;
    subtype Rack_Count is Integer range 1..3;

    type Oven_Type is private; -- The type for the Oven Class

    -- Oven Class Attributes
    -----
    function Energy_Source (Oven : in Oven_Type) return Energy_Type;
    -- Returns the type of energy used to heat the oven

    function Size (Oven : in Oven_Type) return Cubic_Inches;
    -- Returns the number of cubic inches in the oven

    function Temperature (Oven : in Oven_Type) return Degrees;
    -- Returns the current temperature of the oven

    function Num_Racks (Oven : in Oven_Type) return Rack_Count;
    -- Returns the number of racks in the oven

    -- Oven Class Operations
    -----
    procedure Turn_On (Oven : in out Oven_Type);
    -- Turns the oven on

    procedure Turn_Off (Oven : in out Oven_Type);
    -- Turns the oven off

    procedure Set_Temperature (Oven : in out Oven_Type;
                               Temperature : in Degrees);
    -- Sets the desired temperature of the oven

private
    -- The details of Oven_Type go here
end Oven_Class;
```

The body for package `Oven_Class` contains the body for the functions and procedures declared in the package specification. We use functional decomposition to design these subprogram bodies. Any lower level modules developed in the design may be coded as subprograms in the package body. These additional subprograms are encapsulated in the package body and thus not available outside of this body.

The specification of package `Oven_Class` declares `Oven_Type` as `private`. You may not be familiar with this type. We discuss it in detail in Chapter 3. Private types

provide another form of encapsulation. The details of the type are inaccessible to the user of the type. We can declare variables of type `Oven_Type` and pass these variables to the subprograms declared in package `Oven_Class`. We cannot access the details (such as individual fields of a record) of this type. Lack of access to these details does not prevent us from using the type. In fact, as we showed earlier, access to details often makes using an object (like a car's brakes) more difficult. Given the package `Oven_Class` and your current knowledge of Ada, you would have little difficulty in writing a program to turn on an oven, set the desired temperature to 350 degrees, and display a message when the oven reaches that temperature. All of the necessary operations are already defined for you.

When designing the operations of a class, we frequently discover that some additional object classes are needed. When developing these new classes we may find a need for still more classes. Like functional decomposition, object-oriented design makes use of levels of abstraction to control the complexity of the design process. However, the levels in an object-oriented design are based on objects rather than actions (procedures and functions).

Comparing Algorithms

There is more than one way to solve most problems. If you were asked for directions to Joe's Diner (see Figure 1-6), you could give either of two equally correct answers:

1. "Go east on the big highway to the Y'all Come Inn, and turn left," or
2. "Take the winding country road to Honeysuckle Lodge, and turn right."

The two answers are not the same, but because following either route gets the traveler to Joe's Diner, both answers are *functionally correct*.

If the request for directions contained special requirements, one solution might then be preferable to the other. For instance, "I'm late for dinner. What's the quickest route to Joe's Diner?" calls for the first answer, whereas "Is there a pretty road that I can take to get to Joe's Diner?" suggests the second. If no special requirements are known, the choice is a matter of personal preference—which road do you like better?

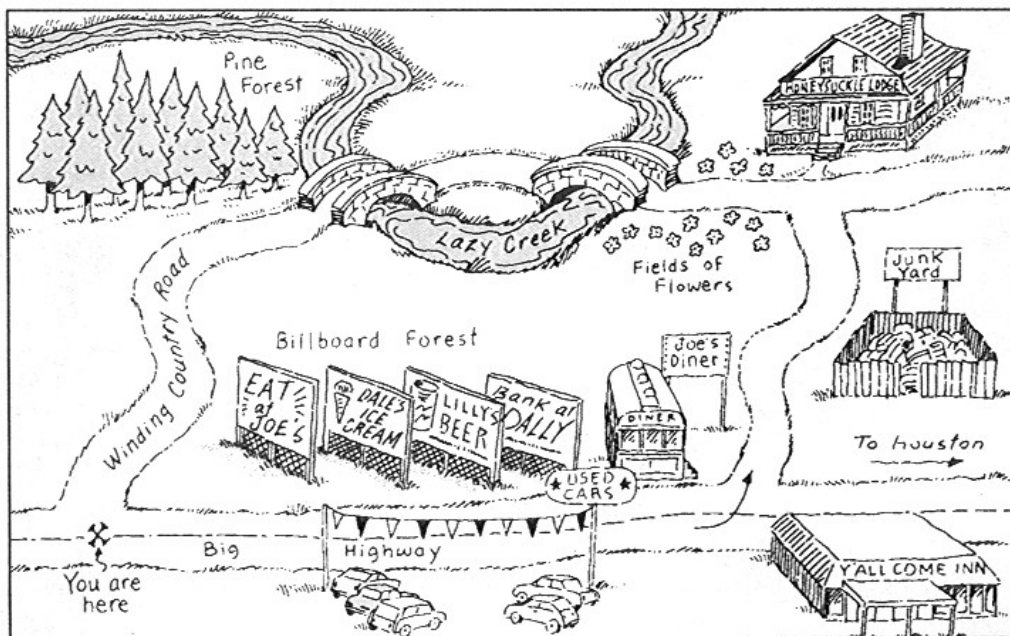


Figure 1-6
Map to Joe's Diner

Algorithm

A logical sequence of discrete steps that describes a complete solution to a given problem in a finite amount of time.

In this book we present numerous algorithms. As you probably already know, an **algorithm** is a step-by-step description of the solution to a problem. All programs, whether designed using functional decomposition or object-oriented techniques contain algorithms. How we choose between two algorithms that do the same task often depends on the requirements of a particular application. If no relevant requirements exist, the choice may be based on the programmer's own style.

Often the choice between algorithms comes down to a question of efficiency: Which one takes the least amount of computing time? Which one does the job with the least amount of work? We are talking here of the amount of work that the *computer* does. Later we also compare algorithms in regard to how much work the *programmer* does. (One is often minimized at the expense of the other.)

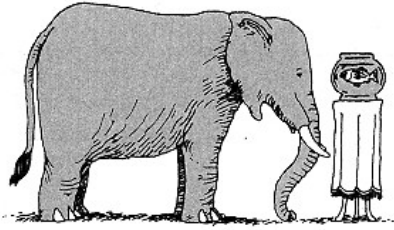
To compare the work done by competing algorithms, we must first define a set of objective measures that can be applied to each algorithm. The *analysis of algorithms* is an important area of theoretical computer science; in advanced courses you will undoubtedly see extensive work in this area. This text discusses a small part of this topic, enough to let you determine which of two algorithms requires less work to accomplish a particular task.

How do programmers measure the work that two algorithms perform? The first solution that comes to mind is simply to code the algorithms and then compare the execution times for running the two programs. The one with the shortest execution time is clearly the better algorithm. Or is it? Using this technique, we can really determine only that Program A is more efficient than Program B *on a particular computer*. Execution times are specific to a particular computer. Of course, we could test the algorithms on all possible computers, but we want a more general measure.

A second possibility is to count the number of instructions or statements executed. This measure, however, varies with the programming language used, as well as with the style of the individual programmer. To standardize this measure somewhat, we could count the number of passes through a critical loop in the algorithm. If each iteration involves a constant amount of work, this measure gives us a meaningful yardstick of efficiency.

These musings lead to the idea of isolating a particular operation fundamental to the algorithm and counting the number of times that this operation is performed. Suppose, for example, that we are summing the elements in an integer array. To measure the amount of work required, we could count the integer addition operations. For an array of 100 elements, there are 99 addition operations. Note, however, that we do not actually have to count the number of addition operations; it is some *function* of the number of elements (N) in the array. Therefore, we can express the number of addition operations in terms of N : For an array of N elements, there are $N - 1$ addition operations. Now we can compare the algorithms for the general case, not just for a specific array size.

If we wanted to compare algorithms for multiplying two float matrices together, we could come up with a measure that combines the float multiplication and addition operations required for matrix multiplication. This example brings up an interesting consideration: Sometimes an operation so dominates the algorithm that the other operations fade into the background "noise." If we want to buy elephants and goldfish, for example, and we are considering two pet suppliers, we really only need to compare the prices of elephants; the cost of the goldfish is trivial in comparison. Similarly, on many computers floating-point multiplication is so much more expensive than addition in terms of computer time that the addition operation is a trivial factor in the efficiency of the whole matrix multiplication algorithm; we might as well count only the multiplication operations, ignoring the addition. In analyzing algorithms, we can often find one operation that dominates the algorithm, effectively relegating the others to the "noise" level.



Big-O

Order of Magnitude (Big-O)

Ways of expressing relationships between large numbers by using formal approximation. Used in computing to express amount of work done.

We have been talking about work as a function of the size of the input to the operation (for instance, the number of elements in the array to be summed). We can express an approximation of this function using a mathematical notation called **order of magnitude**, or **Big-O**, notation. (This is pronounced Big-Oh.) The order of magnitude of a function is identified with the term in the function that increases fastest relative to the size of the problem. For instance, if

$$f(N) = N^4 + 100N^2 + 10N + 50,$$

then $f(N)$ is order N^4 —or, in Big-O notation, $O(N^4)$. That is, for large values of N , N^4 dominates the function.

How is it that we can just drop the low-order terms? Remember the elephants and goldfish that we talked about earlier? The elephants were so much bigger that we could just ignore the goldfish. Similarly, for large values of N , N^4 is so much larger than 50, $10N$, or even $100N^2$ that we can ignore these other terms. This doesn't mean that the other terms do not contribute to the computing time; it only means that they are not significant in our approximation when N is "large."

What is this value N ? N represents the size of the problem. Most of the problems in this book involve data structures—sets, stacks, queues, lists, trees. Each structure is composed of elements. We might develop algorithms to add an element to the structure, or to modify or delete an element from the structure. We can describe the work done by these operations in terms of N , the number of elements in the structure.

Suppose that we want to write all the elements in a list into a file; how much work is that? The answer depends on how many elements are in the list. Our algorithm is

Write to File

```
Create the file
while more elements in list loop
    Put the next element into the file
end loop
```

If N is the number of elements in the list, the "time" required to do this task is

$$(N * \text{time-to-put-one-element}) + \text{time-to-create-the-file}$$

This algorithm is $O(N)$ because the time required to perform the task is proportional to the number of elements (N)—plus a little to create the file. How can we just ignore the create time in determining the Big-O approximation? Assuming that the time necessary to create

a file is constant, this part of the algorithm is our goldfish. If the list only has a few elements, the time needed to open the file may seem significant, but for large values of N , putting the elements into the file is an elephant in comparison to creating the file.

The order of magnitude of an algorithm does *not* tell you how long in microseconds the solution takes to run on your computer. Sometimes we do need that kind of information. For instance, a word processor's requirements state that the program must be able to spell-check a 50-page document (on a particular computer) in less than 120 seconds. For information like this, we do not use Big-O analysis; we use other measurements. In later chapters we compare different implementations of a data structure by coding them and then running a test, recording the time on the computer's clock before and after. This kind of "benchmark" test tells us how long the operations take on a particular computer, using a particular compiler. The Big-O analysis, however, allows us to compare algorithms without reference to these factors.

Some Common Orders of Magnitude

A constant computing time is referred to as $O(1)$. *Constant time* means that the time required for the operation is not dependent on the size of the problem. Assigning a value to the i th element in an array of N elements is $O(1)$, because an element in an array can be accessed directly through its index.

An $O(N)$ algorithm is said to execute in *linear time*. Printing all the elements in an array of N elements is $O(N)$. Searching for a particular value in an array of unordered elements is also $O(N)$, because you (potentially) must search every element in the whole array to find it.

An $O(\log_2 N)$ algorithm does more work than an $O(1)$ algorithm, but less work than an $O(N)$ algorithm. Finding a value in an array of ordered elements using the binary search algorithm is $O(\log_2 N)$. In this search algorithm, the size of the problem (the list) is cut in half with each test. Such halving behavior is usually indicative of an $O(\log_2 N)$ algorithm. (We discuss this algorithm in detail in Chapters 2 and 15.)

A *quadratic time*, or $O(N^2)$, algorithm does more work than an $O(N)$ algorithm when N is large. Most simple sorting algorithms (used to put a list of elements in order) are $O(N^2)$ algorithms. A number of "fast" sorting algorithms are $O(N \log_2 N)$. The table that follows illustrates the relative rates of growth of N^2 and $N \log_2 N$. We discuss this subject at length in Chapter 14.

Comparison of Rates of Growth

N	$\log_2 N$	$N \log_2 N$	N^2	N^3	2^N
1	0	1	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4,096	65,536
32	5	160	1,024	32,768	4,294,967,296
64	6	384	4,096	262,144	About 5 years' worth of instructions on a super computer.
128	7	896	16,384	2,097,152	
256	8	2,048	65,536	16,777,216	Don't ask!

$O(N^3)$ is called *cubic time*. An example of an $O(N^3)$ algorithm is a routine that increments every element in an $N \times N \times N$ three-dimensional array of integers.

Exponential time, or $O(2^N)$, algorithms are really costly. As you can see in the table on page 23, Comparison of Rates of Growth, exponential times increase very dramatically in relation to the size of N . (It is also interesting to note that the values in the last column grow so quickly that the computation time required for problems of this order may exceed the estimated life span of the universe!)

Family Laundry

How long does it take to do a family's weekly laundry? We might describe the answer to this question with the function

$$f(N) = c * N$$

where N represents the number of family members and c is the average number of minutes that each person's laundry takes. We say that this function is $O(N)$ because the total laundry time depends on the number of people in the family. The "constant," c , may vary a little for different families—depending on the size of their washing machine and how fast they can fold clothes, for instance. The time to do the laundry for two different families might be represented with these functions:

$$f(N) = 100 * N$$

$$g(N) = 90 * N$$

But overall, we describe these functions as $O(N)$.

Now what happens if Grandma and Grandpa come to visit the first family for a week or two? The laundry time function becomes

$$f(N) = 100 * (N + 2)$$

We still say that the function is $O(N)$. How can that be? Doesn't the laundry for two extra people take any time to wash, dry, and fold? Of course it does! If N is small (the family consists of Mother, Father, and Baby), the extra laundry for two people is significant. But as N grows large (the family consists of Mother, Father, 12 kids, and a live-in baby-sitter), the extra laundry for two people doesn't really make much difference. (The family's laundry is the elephant; the guest's laundry is the goldfish.) When we compare algorithms using Big-O, we are concerned with what happens when N is "large."

If we are asking the question "Can we finish the laundry in time to make the 7:05 train?" we want a precise answer. The Big-O analysis doesn't give us this information. It gives us an approximation. So, if $100 * N$, $90 * N$, and $100 * (N + 2)$ are all $O(N)$, how can we say which is "better"? We can't—in Big-O terms, they are all roughly equivalent for large values of N . Can we come up with a better algorithm for getting the laundry done? If the family wins the state lottery, they can drop all their dirty clothes off at a professional laundry 12 minutes' drive from their house (two 24-minute round-trips). Now the function is

$$f(N) = 48$$

This function is $O(1)$. The answer is independent of the number of people in the family. If they switch to a laundry 5 minutes from their house, the function becomes

$$f(N) = 20$$

This function is also $O(1)$. In terms of Big-O, the two professional laundry solutions are equivalent. No matter how many family members or house guests, it takes a constant amount of the family's time to do the laundry. (We aren't concerned with the professional laundry's time.)

Note that throughout this discussion, we have been talking about the amount of work the computer must do to execute a program. This determination does not necessarily relate to the size of the program, say, in lines of code. Consider the following two algorithms to initialize to zero every element in an N -element array.

Algorithm Init1

```
List(1) := 0;
List(2) := 0;
List(3) := 0;
List(4) := 0;
List(5) := 0;
.
.
.
List(N) := 0;
```

Algorithm Init2

```
for Index in 1..N loop
    List(Index) := 0;
end loop;
```

Both algorithms are $O(N)$, even though they greatly differ in the number of lines of code.

Now let's look at two different algorithms that calculate the sum of the integers from 1 to N . Algorithm Sum1 is a simple `for` loop that adds successive integers to keep a running total:

Algorithm Sum1

```
Sum := 0;
for Count in 1..N loop
    Sum := Sum + Count;
end loop;
```

That seems simple enough. The second algorithm calculates the sum by using a formula. To understand the formula, consider the following calculation when $N = 9$.

$$\begin{array}{cccccccccc}
 1 & + & 2 & + & 3 & + & 4 & + & 5 & + & 6 & + & 7 & + & 8 & + & 9 \\
 + & 9 & + & 8 & + & 7 & + & 6 & + & 5 & + & 4 & + & 3 & + & 2 & + & 1 \\
 \hline
 10 & + & 10 & + & 10 & + & 10 & + & 10 & + & 10 & + & 10 & + & 10 & + & 10 & = & 10 * 9 = 90
 \end{array}$$

We pair up each number from 1 to N with another, such that each pair adds up to $N + 1$. There are N such pairs, giving us a total of $(N + 1) * N$. Now, since each number is included twice, we simply divide the product by 2. Using this formula, we can solve the problem: $((9 + 1) * 9)/2 = 45$. Now we have a second algorithm:*

* You can use proof by induction to show that this formula works for all numbers.

Algorithm Sum2

```
Sum: = ( (N + 1) * N ) / 2;
```

Both of the algorithms are short pieces of code. Let's compare them using Big-O notation. The work done by Sum1 is a function of the magnitude of N ; as N gets larger, the amount of work grows proportionally. If N is 50, Sum1 works 10 times as hard as when N is 5. Algorithm Sum1, therefore, is $O(N)$.

To analyze Sum2, consider the cases when $N = 5$ and when $N = 50$. They should take the same amount of time. In fact, whatever value we assign to N , the algorithm does the same amount of work to solve the problem. Algorithm Sum2, therefore, is $O(1)$.

Does that mean that Sum2 is always faster? Is it always a better choice than Sum1? That depends. Sum2 might seem to do more "work" since the formula involves multiplication and division, while Sum1 is a simple running total. In fact, for very small values of N , Sum2 might actually do more work than Sum1. Of course, for very large values of N , Sum1 does a proportionally larger amount of work, while Sum2 stays the same. So the choice between the algorithms depends in part on how they are used, for small or large values of N .

Another issue is the fact that Sum2 is not as obvious as Sum1; thus it is harder for the programmer (a human) to understand. Sometimes a more efficient solution to a problem is more complicated; we may save computer time at the expense of the programmer's time. Of course, good documentation of a complicated solution may save the *next* programmer a good deal of time.

What's the verdict? As usual in the design of computer programs, there are trade-offs. We must look at our program's requirements and then decide which solution is better. Throughout this text we examine different choices of algorithms and data structures. We compare them using Big-O, but we also examine the program's requirements and the "elegance" of the competing solutions. As programmers, we design software solutions with many factors in mind.

Summary

How are our quality software goals met by the strategy of abstraction and information hiding? When details are hidden at each level, the code becomes simpler and more readable, which in turn makes the program easier to write and modify. Both functional decomposition and object-oriented design processes produce modular units that are also easier to test, debug, and maintain.

One positive side effect of modular design is that modifications tend to be localized in a small set of modules, and thus the cost of modifications is reduced. Remember that whenever a module is modified it must be retested to make sure that it still works correctly in the program. By localizing the modules affected by changes to the program, we limit the extent of retesting needed.

Finally, reliability is increased by making the design conform to our logical picture and delegating confusing details to lower levels of abstraction. An understanding of the wide range of activities involved in software development—from requirements analysis through the maintenance of the resulting program—leads to an appreciation of a disci-

plined software engineering approach. Everyone knows some programming wizard who can sit down and hack out a program in an evening, working alone, coding without a formal design. But we cannot depend on wizardry to control the design, implementation, verification, and maintenance of large, complex software projects that involve the efforts of many programmers. As computers grow larger and more powerful, the problems that people want to solve on them also become larger and more complex. Some people refer to this situation as a software *crisis*. We'd like you to think of it as a software *challenge*.

Exercises

1. Explain what is meant by "software engineering."
2. Which of these statements is always true?
 - (a) All of the program requirements must be completely defined before design begins.
 - (b) All of the program design must be complete before any coding begins.
 - (c) All of the coding must be complete before any testing can begin.
 - (d) Different development activities often take place concurrently, overlapping in the software life cycle.
3. Name three computer hardware tools that you have used.
4. Name two software tools that you have used in developing computer programs.
5. Explain what is meant in this chapter by "ideaware."
6. Goal 1, "Quality software works," means that the program meets its _____, as documented in the software _____.
7. Software specifications tell exactly _____ a program does, but not _____.
8. Name three program elements that you might see detailed in a software specification.
9. Name two ways in which you can make your programs meet Goal 2, "Quality software can be read and understood."
10. Explain why software might need to be modified
 - (a) in the design phase
 - (b) in the coding phase
 - (c) in the testing phase
 - (d) in the maintenance phase
11. Goal 4 says, "Quality software is completed on time and within budget."
 - (a) Explain some of the consequences of not meeting this goal for a student preparing a class programming assignment.
 - (b) Explain some of the consequences of not meeting this goal for a team developing a highly competitive new software product.
 - (c) Explain some of the consequences of not meeting this goal for a programmer who is developing the user interface (the screen input/output) for a spacecraft launch system.
12. What is the first step in developing any software?
13. You are working on a class programming assignment, and the details of one of the requirements are ambiguous. What are some ways of dealing with this problem?
14. Write two additional scenarios for the automated teller machine described in this chapter. One of your scenarios should include an abnormal event (such as bad PIN or insufficient funds).

15. For each of the following, describe at least two different abstractions for different viewers (see Figure 1-2):

(a) a dress	(d) a key
(b) an aspirin	(e) a saxophone
(c) a carrot	(f) a piece of wood

16. For each of the following, draw a "part of hierarchy (see Figure 1-3).

(a) a house	(d) a microwave oven
(b) a car	(e) a saxophone
(c) a farm	(f) a university

17. Draw "kind of" hierarchies (see Figure 1-4) with the following at the top:

(a) building	(d) cooking utensil
(b) vehicle	(e) musical instrument
(c) geometric figure	(f) music

18. Explain what is meant by "information hiding." Why is the following section of code an example of information hiding?

```
Tax := Rate * Taxable_Income;
Text_IO.Put ("Your tax liability is");
Float_IO.Put (Item => Tax, Fore => 6, Aft => 2, Exp => 0);
```

19. Functional decomposition is based on a hierarchy of _____ whereas object-oriented design is based on a hierarchy of _____.

20. What is the difference between an object and an object class? Give some examples.

21. Make a list of potential objects from the description of the automated teller machine scenario given in this chapter.

22. Object classes have _____ and _____ associated with them.

23. (a) Explain what is meant by "encapsulation."

(b) Give an example (other than a watch case) of encapsulation in everyday life.

(c) What is Ada's major mechanism for encapsulation?

24. Write a list of oven attributes that are *not* relevant to cooking problems. Such attributes would not be included in a package specification for an oven class.

25. What is meant by "implementation"?

26. What is an Ada package?

27. Why are Ada packages written in two parts?

28. *True or false?* There is usually one way of solving a problem that is clearly better than any other.

29. Give an example of an algorithm (other than the examples discussed in the chapter) that is

(a) $O(1)$	(b) $O(N)$	(c) $O(N^2)$

30. A routine to calculate the sum of the square roots of some values in array `Data` contains the following code segment:

```
Sum_of_Sqr_Rt := 0.0;
Index := 1;
loop
  exit when Index > Num_Elements;
  SR := Square_Root(Value => Data(Index));
  Sum_of_Sqr_Rt := Sum_of_Sqr_Rt + SR;
  Index := Index + 1
end loop;
```

- (a) Identify the "goldfish" and the "elephant" operations inside the loop.
- (b) If the function `Square_Root(X)` is $O(X)$, what is the order of the sum-of-square-roots algorithm with respect to `Num_Elements`?
31. Algorithm 1 does a particular task in a "time" of N^3 where N is the number of elements processed. Algorithm 2 does the same task in a "time" of $3N + 1000$.
- (a) What are the Big-O requirements of each algorithm?
- (b) Which algorithm is more efficient by Big-O standards?
- (c) Under what conditions, if any, would the "less efficient" algorithm execute more quickly than the "more efficient" algorithm?
32. Three algorithms do the same task. Algorithm 1 is $O(N)$, Algorithm 2 is $O(N)$, and Algorithm 3 is $O(\log_2 N)$. Which algorithm should execute the fastest for large values of N ? Which one should execute the slowest?
33. A good way to understand the differences in orders of magnitudes for the Big-O notation is to graph functions of the same magnitude. For example, $O(1)$ would be represented, by the function $f(x) = 1$, $O(N)$ would be represented by the function $f(x) = x$, and $O(N^2)$ would be represented by the function $f(x) = x^2$. Graph each of these functions on the same coordinate system and compare the rates at which they increase for large values of x .
34. Your professor hands out a programming assignment that is supposed to be turned in at a Tuesday morning class two weeks from today. Your sister's wedding is the weekend before your program is due. Make up a schedule that allows you to get your program done without ruining the wedding weekend.

Programming Problems

- Design and implement an Ada program that meets the specification of Figure 1-1.
- You are the manager of a team of ten programmers who have just completed a seminar in software engineering. To prove to your boss that these techniques pay off, you decide to run the following contest: You number the programmers 1 through 10, based on their performance in the seminar (1 is poorest, 10 is best) and monitor their work. As each does his or her part of your project, you keep track of the number of lines of debugged code turned in by each programmer. You record this number as a programmer turns in a debugged module. The winner of the contest is the first person to reach 1000 lines of debugged code. (You hope this is programmer 9 or 10.) As further proof of the value of these new techniques, you want to determine how many poor programmers it takes to surpass the winner's figure; that is, find the smallest k such that programmers 1 through k have turned in more lines than the winner.

Input

The input consists of a sequence of pairs of integers. The first integer in each pair is the programmer's number (an integer from 1 to 10), and the second is the number of lines of code turned in. The pairs occur in the same order as that in which the modules were turned in.

Processing/Output

Read in pairs of integers until someone's total goes over 1000. Print out (echo print) each pair as you read it. Ignore any input after someone's total exceeds 1000. Then print out a table listing the ten programmers and their totals, with the winner flagged as shown in the example on page 30. Finally, find the smallest k such that the sum of the totals for programmers 1 - k exceeds the winner's total. Print k in an explanatory sentence.

Sample Input

```

10 230
8 206
7 111
3 159
9 336
1 51
10 250
4 101
9 341
2 105
8 256
10 320
3 150
5 215
7 222
9 400      9 goes over 1000.

```

Sample Output

```

      PROGRAMMER PROGRESS
Programmer      Lines of Code
    10              230      Echo print the first 16 pairs.
    .
    .
    9              400

      FINAL TOTALS
Programmer      Lines of Code
    1              51
    2              105
    3              309
    4              101
    5              215
    6              0
    7              333
    8              462
    9             1077      *** THE WINNER ***
   10              800
It took programmers 1 through 7 to produce more than the winner.

```

2—

Verifying, Testing, and Debugging

GOALS

After reading this chapter, you should be able to

- identify several sources of program errors and describe strategies to prevent software errors
- specify the preconditions and postconditions of a program segment or subprogram
- use loop invariants to minimize loop design errors
- describe how deskchecking, code walk-throughs, and design and code inspections can improve software quality and reduce software effort
- state testing goals and describe testing strategies

In Chapter 1 we discussed some characteristics of good programs. The first of these was that a good program works—it accomplishes its intended function. How do you know when your program meets that goal? The simple answer is, test it.

Let's look at testing in perspective with the rest of the software development process. As programmers, we first make sure that we understand the requirements, and we come up with a general solution. Then we design the solution to the program, using good design principles. Next we implement the solution, using good structured coding, with packages, procedures, functions, self-documenting code, and so on.

Once we have the program coded, we compile it repeatedly until there are no more syntax errors. Then we run the program and see what happens. If the program doesn't work, we say that it has a "bug" in it. We try to pinpoint the error and fix it, a process called *debugging*. When all the bugs are out, the program is correct.

Well, maybe it's correct. Testing is useful in revealing the presence of bugs in a program, but it doesn't prove their absence. We can only say for sure that the program worked correctly for the cases we tested. This approach seems somewhat haphazard. How do we know which tests or how many of them to run? Testing a whole program at once isn't easy. And fixing the errors found during such testing sometimes can be a messy task. Too bad we couldn't have detected the errors earlier—while we were designing the program, for instance. They would have been much easier to fix then.

In the previous chapter we talked about ways to do a good job on the design and implementation part of the programming process. We know how program design can be improved by using a good design methodology. Is there something similar that we can do to improve our program verification activities?

Program Verification

The process of determining the degree to which a software product fulfills its specifications.

Yes, there is. **Program verification** activities don't need to start when the program is completely coded; they can be incorporated into the whole software development process, from the requirements phase on. Program verification is more than just testing.

Can we really debug a program before it has ever been run—or even before it has been written? In this chapter we discuss a number of topics related to satisfying our Programming Goal 1: "Quality software works." The topics include

- designing for correctness
- performing code and design walk-throughs and inspections
- choosing test goals and data
- writing test plans
- structured integration testing
- systematic debugging

Where Do Bugs Come From?

When Sherlock Holmes goes off to solve a case, he doesn't start from scratch every time; he knows from experience all kinds of things that help him find solutions. Suppose Sherlock Holmes finds a victim in a muddy field. He immediately looks for footprints in the mud, for he can tell from a footprint what kind of shoe made it. The first print he finds matches the shoes of the victim, so he keeps looking. Now he finds another, and from his vast knowledge of footprints he can tell that it was made by a certain type of boot. He deduces that such a boot would be worn by a particular type of laborer, and from the size and depth of the print he guesses the suspect's height and weight. Now, knowing some-



thing about the habits of laborers in this town, he guesses that at 6:30 P.M. the suspect might be found in Clancy's Pub.

In software verification we often are expected to play detective. Given certain clues, we have to find the bugs in programs. If we know what kinds of situations produce program errors, we are more likely to be able to detect and correct problems. We may even be able to step in and prevent many errors entirely, just as Sherlock Holmes sometimes intervenes in time to prevent a crime that is about to take place.

Let's look at some types of software errors that show up at various points in program development and testing, and see how they might be avoided.

Errors in the Specifications and Design

What would happen if, shortly before you were supposed to turn in a major class assignment, you discovered that some details in the professor's program description were incorrect? To make matters worse, you found out that the corrections were discussed at the beginning of class on the day you got there late. But somehow you never knew about the problem until your tests of the class data set came up with the wrong answers. What do you do now?

Writing a program to the wrong specifications is probably the worst kind of software error. How bad can it be? Let's look at a true story. Some time ago, a computer company contracted to replace a government agency's obsolete system with new hardware and software. The company wrote a large and complicated program, based on specifications and algorithms provided by the customer. The developers checked the new system at every point in its development to ensure that its functions matched the requirements in the specifications document. When the system was complete and the new software was executed, they discovered that the results of its calculations did not match those of the old system. A careful comparison of the two systems showed that the specifications of the new software were erroneous because they were based on algorithms taken from the old system's inaccurate documentation. The new program was "correct" because it accomplished its specified functions, but the program was useless to the customer because it didn't accomplish its intended functions—it didn't work. The cost of correcting the errors measured in the millions of dollars.

How could correcting the error be so expensive? First of all, much of the conceptual and design effort, as well as the coding, was wasted. It took a great deal of time to pinpoint which parts of the specification were in error and then to correct this document before the

program could be redesigned. Then much of the software development activity (design, coding, and testing) had to be repeated.

This case is an extreme one, but it illustrates how critical specifications are to the software process. In general, programmers are more expert in software development techniques than in the "application" areas of their programs, such as banking, city planning, satellite control, or medical research. Thus correct program specifications are crucial to the success of program development.

This case also illustrates a basic principle about software costs: The earlier in the development cycle a problem is detected, the cheaper it is to fix. "Cost" may mean dollars to pay programmers, monetary penalties for missed schedules, or points off for turning in a course assignment late. Because the development of the specifications for a software assignment precedes its design and implementation, an undetected error at this point can be very expensive. The longer the problem goes without detection, the higher the cost of fixing it rises.

Figure 2-1 shows how fast the costs rise in subsequent phases of software development. The vertical axis represents the relative cost of fixing an error; this cost might be in units of hours, or hundreds of dollars, or "programmer months" (the amount of work one programmer can do in a month). The horizontal axis represents the stages in the development of a software product. As you can see, an error that would have taken one unit to fix when you first started designing might take a hundred units to correct when the product is actually in operation!

Many specification errors can be prevented by good communication between the programmers (you) and the party who originated the problem (the professor, manager, or customer). It always pays to ask questions when you don't understand something in the program specifications. And the earlier you ask, the better.

A number of questions may come to mind as you first read a programming assignment. What error checking is necessary? What algorithm or data structure is supposed to be used in the solution? What assumptions are reasonable? If you obtain answers to these questions when you first begin working on an assignment, you can incorporate them into

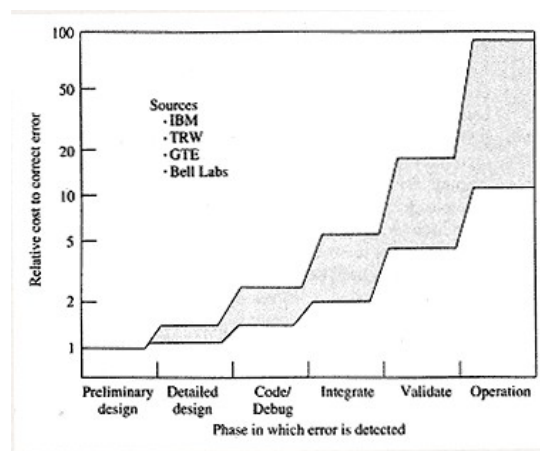


Figure 2-1
A Graph to Demonstrate the Importance of Early Detection of Software Errors

your design and implementation of the program. Later in the program's development, unexpected answers to these questions can cost you time and effort. In short, in order to write a program that is correct, you must understand precisely what it is that your program is supposed to do. Learning to ask the right questions comes with experience gained by writing software. Reading case studies such as those presented in this book is another way to help develop these necessary skills.

Sometimes specifications change during the design or implementation of a program. In this case, a good design helps you to pinpoint which sections of the program have to be redone. The parts of a program that require changes usually can be located more easily from a design than from the code itself.

Compile-Time Errors

In the process of learning Ada in your first programming course, you probably made a number of syntax errors. These resulted in error messages (for example, TYPE MISMATCH, ILLEGAL ASSIGNMENT, MISSING END, and so on) when you tried to compile the program. That's par for the first course.

Now that you are more familiar with the programming language, you can save your debugging skills for tracking down really juicy logical errors. Try to get the syntax right the first time. Having your program compile cleanly on the first attempt is not an unreasonable goal. A syntax error wastes computing time and money, as well as programmer time, and it is preventable.

Some programmers argue that looking for syntax errors is a waste of time, that it is faster to let the compiler catch all your typos and syntax errors. Sometimes, however, a coding error turns out to be a legal statement, syntactically correct but semantically wrong. This situation may cause very obscure, hard-to-locate errors.

Programmers must be familiar with all the idiosyncrasies of the implementation of a programming language at their particular site. For instance, you may be programming in Meridian Ada® on a personal computer at school, whereas in your first job, you may program in DEC Ada® on a workstation. More than likely, you will not even be programming in Ada, for there are literally thousands of computer languages in use. Everyone knows of BASIC, C, and Pascal. You may also have heard of Smalltalk, LISP, C++, and Modula-2. But you may find yourself programming in SNOBOL, Prolog, Eiffel, or MUMPS.

Learning the programming language is often the easiest part of a new software assignment. This does not mean, however, that the language is the least important part. In this book we discuss abstract data types and algorithms that we claim are language independent. This means that they can be implemented in almost any general-purpose programming language. The success of the implementation, however, is dependent on a thorough understanding of the features of the programming language. What is considered good programming practice in one language may be poor programming practice in another.

For instance, suppose a program uses a list of data objects, each of which contains several fields. A FORTRAN programmer might implement this abstract data structure as a series of parallel arrays, one array per field. The same implementation would work in Ada, but it's a poor solution; in Ada, an array of records would be more natural. FORTRAN has no records; the FORTRAN programmer must instead use separate arrays.

Sometimes two programming languages have rules and features that look similar but work differently. For instance, the language called Modula-2 has rules for naming identifiers (variables, constants, procedures, and so on) that are similar to Ada's. There is an important difference, however; Modula-2 is case sensitive. Upper and lowercase letters

are considered to be different characters in Modula-2. This means that `DataList`, `datalist`, `DATALIST`, and `Datalist` represent four different Modula-2 variables. What seem to be minor details in a programming language manual may turn out to be very important in practice.

Therefore it's worthwhile to develop an expert knowledge of both the control and data structures and the syntax of the language in which you are programming. In general, if you have a good knowledge of your programming language, and are careful, you can avoid syntax errors. The ones you might miss are relatively easy to locate and correct. Most are flagged by the compiler with an error message. Once you have a "clean" compilation, you can link and execute your program.

Run-Time Errors

Errors that occur during the execution of a program are usually harder to detect than syntax errors. Some run-time errors stop execution of the program. We say that the program "crashed" or "abnormally terminated."

Run-time errors are often the result of the programmer making too many assumptions. For instance,

```
Result := Dividend / Divisor;
```

is a legitimate assignment statement, if we can assume that `Divisor` is never zero. However, if `Divisor` is zero, a run-time error results.

Ada automatically makes many checks for run-time errors, so we could just passively let the system handle any exceptions it finds. However, Ada usually takes care of such errors by terminating the program, and a program that crashes whenever it detects an error condition is not very satisfying to use.

You must not allow conditions that stop your program's execution; your program should stay in control until the user is ready to quit. A program should not crash when it tries to open a file that doesn't exist. So you should include exception handlers for such conditions with appropriate error recovery.

Robustness

The ability of a program to recover following an error; the ability of a program to continue to operate within its environment.

The ability of a program to recover when an error occurs is called **robustness**. If a commercial program is not robust, people will not buy it. Who wants a word processor that crashes if the user says "SAVE" when there is no disk in the drive? We want the program to tell us, "Put your disk in the drive, and hit Enter." For some types of software, robustness is a critical requirement. Programs like an airplane's automatic pilot system or an intensive care unit's patient-monitoring program just can't afford to crash. These are situations in which defensive posture produces good results.

In general, you should include active checks or exception handlers for error-creating conditions rather than let them abort your program. For instance, it's generally unwise to make too many assumptions about the correctness of input, especially "interactive" input from a keyboard. A better approach is to provide `DATA_ERROR` and `CONSTRAINT_ERROR` exception handlers to ensure the correct type and bounds of such input. The programmer then can decide how an error should be handled (request new input, print a message, or go on to the next data) rather than leave the decision to the system. Even the decision to quit should be made by a program that is in control of its own execution. If worse comes to worst, let your program die gracefully.

This does not mean that *everything* that the program inputs must be checked for errors. Sometimes inputs are known to be correct—for instance, input from a file produced

by another program. The decision to include error checking must be based upon the requirements of the program.

Some run-time errors do not stop execution but produce the wrong results. You may have incorrectly implemented an algorithm or used a variable before it was assigned a value. You may have inadvertently swapped two parameters of the same type on a procedure call. These "logical" errors are often the hardest to prevent and to locate. Later in this chapter we talk about debugging techniques to help pinpoint run-time errors. We also discuss structured testing methods that isolate the part of the program being tested. But knowing that the earlier we find an error the easier it is to fix, we turn now to ways of catching run-time errors before run time.

Designing for Correctness

It would be nice if there were some tool that would locate the errors in our design or code without our having to even run the program. That sounds unlikely, but consider an analogy from geometry. We wouldn't try to prove the Pythagorean Theorem by proving that it worked on every triangle; that would only demonstrate that the theorem works for every triangle we tried. We prove theorems in geometry mathematically. Why can't we do the same for computer programs?

The verification of program correctness, independent of data testing, is an important area of theoretical computer science research. The goal of this research is to establish a method for proving programs correct that is analogous to the method for proving theorems in geometry. The necessary techniques exist, but the proofs are often more complicated than the programs themselves. Therefore a major focus of verification research is the attempt to build automated program provers—verifiable programs that verify other programs. Until this research is complete, the formal verification techniques can be carried out by hand, to prove logically that pieces of a program are correct.

Assertions and Program Design

In this chapter we consider a couple of ideas upon which verification techniques are based, because these concepts can help us in our effort to design correctness into our programs. These concepts involve types of assertions that we can make about what our software is trying to do. An *assertion* is a logical proposition that can be true or false. We can make assertions about the state of the program. For instance, following the assignment statement

```
Sum := Part + 1;    -- Sum and Part are integers
```

we might assert: "The value of `Sum` is greater than the value of `Part`." That assertion might not be very useful or interesting by itself, but let's see what we can do with it. We can demonstrate that the assertion is true by making a logical argument: No matter what value `Part` has (negative, zero, or positive), when it is increased by 1, the result is a larger value. Now note what we *didn't* do. We didn't have to run a program containing this assignment statement to verify that the assertion was correct.

The general concept behind formal program verification is that we can make assertions about what the program is intended to do, based on its specifications, and then prove through a logical argument (rather than through execution of the program) that a design or implementation satisfies the assertions. Thus we can break the process down into two

steps: (1) correctly asserting the intended function of the part of the program to be verified and (2) proving that the actual design or implementation does what we asserted. The first step, making assertions, sounds as if it might be useful to us in the process of designing correct programs. After all, we already know that we cannot write correct programs unless we know what they are supposed to do.

Preconditions and Postconditions

Let's take the idea of making assertions down a level in the design process. Suppose we want to design a specific operation. To ensure that this operation fits into the program as a whole, we must clarify what happens at its *boundaries*—what must be true before we use the operation and what is true when we come out.

Preconditions

Assertions that must be true on entry into an operation or procedure for the postconditions to be guaranteed.

To make the task more concrete, picture the operation as it will eventually be coded, as a procedure that will be called within a program. To be able to call the procedure, we must know its exact interface: the name and the parameter list, which indicates its inputs and outputs. But this isn't enough. We also must know any assumptions that must be true for the operation to function correctly. We call the assertions that must be true on entry into the procedure **preconditions**. The preconditions are like a product disclaimer:

WARNING: If you try to execute this operation when the preconditions are not true, the results are not guaranteed.

For instance, when we said on page 37 that following the execution of

```
Sum := Part + 1;
```

we can assert that `Sum` is greater than `Part`, there was an assumption—a precondition—that `Part` is not `Integer ' Last`. If this precondition were violated, our assertion would not be true.

Postconditions

Assertions that state what results are to be expected at the exit of an operation or procedure, assuming that the preconditions are true.

We also must know what conditions are true when the operation is complete. The **postconditions** are assertions that describe the results of the operation. The postconditions do not tell us how these results are accomplished; they merely tell us what the results should be.

Let's take a look at what the preconditions and postconditions might be for a simple operation: a procedure that deletes the last element from a list and returns its value as an output.

Get_Last (List: in out; Last_Value: out)

Function: Remove the last element from List and return its value in Last_Value.

Preconditions: List is not empty; it contains at least one element.

Postconditions: Last_Value = value of last element from List;
List = original List minus its last element.

What do these preconditions and postconditions have to do with program verification? By making explicit assertions about what is expected at the operation interfaces, we can avoid making logical errors based on misunderstandings. For instance, from the precondition we know that we must check outside of this operation for the empty condition; this module *assumes* that there is at least one element. The postcondition tells us that when the

value of the last list element is retrieved, that element is deleted from the list. This fact is an important one for the list user to know. If we just want to take a peek at the last value without affecting the list, we cannot use `Get_Last`.

Experienced software developers will tell you that misunderstandings about interfaces to someone else's operations are one of the main sources of program problems. We continue to use preconditions and postconditions in specifications at the operation level in this book, because the information they provide helps us to design programs in a truly modular fashion. We then can use the objects we've designed in our programs, confident that we are not introducing errors by making mistakes about assumptions and about what the object operations actually do.

Loop Invariants

Taking our design process down a few more levels, we get into the actual control structures of the design: blocks of statements, branches, and loops. Loops are known trouble-makers—sometimes they go on forever and sometimes they don't do what we meant them to do. They also have a bad habit of executing one too many times or one too few times.

Loop Invariant

Assertion of what conditions must be true on entry into an iteration of the loop body and on exit from the loop.

Using an assertion called a loop invariant can help you design error-free pretest loops.* **Loop invariants** are assertions that must be true at the start of every iteration of the loop body and when the loop terminates. (The loop invariant must always be true; that's why it's called an invariant.) The loop invariant is not the same thing as the condition in the while statement which must also be true at the start of each iteration of the loop body. The loop invariant also says something about the purpose and semantics (meaning) of the loop.

Let's look at an example. Procedure `Sum_Values` has the following specification:

Sum Values (List, Num_Values, Sum)

Function: Sum all the elements in List.

Input: List (an array), Num_Values (an integer)

Preconditions: List contains at least Num_Values elements
Num_Values < Integer'Last

Output: Sum

Postconditions: Sum = sum of the elements in List(1 .. Num_Values)

Here's the algorithm used by Sum Values:

Sum Values

```
Sum := 0
Index := 1
while Index <= Num_Values loop
  Add List(Index) to Sum
  Increment Index
end loop
```

*We illustrate pretest loop invariants with the while loop statement. Loop invariants can also be applied to pretest loops implemented with the loop and exit statements.

What can we assert must be true on entry into the loop in this algorithm? Of course we can say that `Index` is less than or equal to `Num_Values`; we know that from the condition on the while statement. But we can also assert that, on each entrance into the loop body,

Sum contains the sum of elements in `List (1 .. Index - 1)`.

This assertion is true initially because `Sum` contains 0, which is the sum of the elements in `List (1 .. 0)`. (There are no elements in a null range, and we take the sum of "no elements" to be zero.)

When the loop terminates, with `Index = Num_Values + 1`, the invariant asserts that `Sum` contains the sum of the elements from `List (1)` to `List (Index - 1)` (which is now `List ((Num_Values + 1) - 1)`, or simply `List (Num_Values)`). Thus we can see that the loop is actually accomplishing what we intended, to sum all the elements in `List`.

Another check we should always make when dealing with loops is to see whether the loop ever terminates. Part of the loop invariant deals with the terminating conditions; this is the part that usually shows up on the while statement. This condition (for example, `Index <= Num_Values`) should be considered together with the initialization (`Index := 1`) and its update mechanism within the loop (Increment `Index`). Because `Index` starts out at 1 and is unconditionally incremented inside the loop, we know that `Index` eventually exceeds `Num_Values`. Thus we are certain that the loop terminates.

A correct loop invariant is very helpful in determining what the loop body is intended to accomplish; therefore, it is a good idea to include it as a comment in the code:

```
-- Loop invariant: Index may range from 1 .. Num_Values + 1 AND
--                Sum = sum of elements in List(1 .. Index - 1).
while Index <= Num_Values loop
```

Let's take another look at the relationship between the loop invariant and the Boolean expression that controls the loop (that is, `Index <= Num_Values`). On entrance to each iteration of the loop body,

Loop_Invariant AND Boolean_Expression is True,

while following the execution of the loop,

Loop_Invariant AND NOT Boolean_Expression is True.

Note that the loop invariant is not only true on the entrance into the loop—it is also true on the exit. As we already know, the Boolean expression must be true to enter the loop, and must be false for the loop to terminate.

Now let's write a similar procedure, `Multiply_Ten`, with the following specifications:

Multiply_Ten (List, Num_Values, Product)

Function: Calculate the product of the first 10 nonzero elements in `List`.

Input: `List` (an array), `Num_Values` (an integer)

Preconditions: `List` contains at least `Num_Values` elements.

Output: `Product`

Postconditions: `Product` = the product of the first 10 nonzero elements in `List(1 .. Num_Values)`.

If there are fewer than 10 nonzero elements in `List`, `Product` =
Product of all the nonzero elements in `List(1 .. Num_Values)`.

This procedure seems almost as easy as `Sum_Values`, so we try to write the algorithm right away.

Multiply Ten

```

Product := 1
Index := 1
Non_Zero_Count := 0
while Non_Zero_Count <= 10 loop
  if List(Index) /= 0 then
    Multiply List(Index) by Product
    Increment Non_Zero_Count
  end if
  Increment Index
end loop

```

First of all let's see whether the loop terminates. The terminating condition says `Non_Zero_Count <= 10` (because we want to calculate the product of the first ten nonzero elements). `Non_Zero_Count` starts out at 0 and is incremented whenever the current list element is not 0. But we don't have any guarantees (no preconditions) that the list actually contains ten nonzero elements. If it doesn't, the loop tries to go on looking past the end of the array and raise `CONSTRAINT_ERROR`. (That's one way of terminating a loop!)

Let's try again, this time writing the invariant first. The invariant consists of all the assertions that must be true on entry into the loop body in order for it to work correctly. For instance, what do we know about `Index`? `Index` tells us how far we have processed in the array; therefore its value must be within the range of index values on entrance to the loop. But there's nothing in the loop condition of our algorithm that addresses the issue of range! We have forgotten that we need a condition to stop when `Index` gets to the end of the array. The loop invariant is a good reminder: We can keep entering the loop while there are still elements to process. Here's the first part of the loop invariant: *Index may range from 1 .. Num_Values + 1*.

Why do we say *Num_Values + 1*? Remember that the loop invariant is true on exit from the loop, as well as on entrance to each iteration. The Boolean expression that controls the loop says `while Index <= Num_Values`. On entrance to each iteration, the loop invariant AND the Boolean expression is true. Following the execution of the loop (when `Index = Num_Values + 1`), the Boolean expression is false, and the loop invariant is still true.

This part of the loop invariant, as we have phrased it above, is rather wordy. We express this idea (`Index may range from 1 .. Num_Values + 1`) more concisely as follows:

Loop Invariant Part 1: `1 <= Index <= Num_Values + 1`

What about `Non_Zero_Count`? Before we talk about its range of values, let's ask ourselves what `Non_Zero_Count` means in the loop.

Loop Invariant Part 2: `Non_Zero_Count = the number of nonzero elements in List(1 .. Index - 1)`

When we look at `Non_Zero_Count`'s meaning in the loop invariant, we see that we've made a mistake in its contribution to the while condition. Coming into each iteration of the loop body, `Index` tells us which element is *about* to be processed, so `Non_Zero_Count` is the total number of nonzero elements in `List(1 .. Index - 1)`. `Non_Zero_Count`, however, tells us how many nonzero elements have *already* been processed at the top of

the loop. So when its value is 10 we want to stop instead of reentering the loop body. There's another part of the loop invariant to indicate that `Non_Zero_Count` may range from 0 to 10:

Loop Invariant Part 3: $0 \leq \text{Non\_Zero\_Count} \leq 10$

This part of the loop invariant is true on exit from the loop, as well as on entrance to each iteration of the loop body. The associated Boolean expression, `Non_Zero_Count < 10`, must be true on entrance to each iteration, and is false on exit from the loop (assuming that there are ten nonzero elements in the list).

Now that we understand the meanings of `Index` and `Non_Zero_Count`, we can also write the part of the loop invariant that defines `Product`.

Loop Invariant Part 4: `Product = product of the nonzero elements in List(1 .. Index - 1)`

The resulting condition is

```
while Index <= Num_Values AND Non_Zero_Count < 10 loop
```

The revised algorithm is

Multiply Ten (revised)

```
Product := 1
Index := 1
Non_Zero_Count := 0
Loop Invariant:
    1 <= Index <= Num_Values + 1 AND
    0 <= Non_Zero_Count <= 10 AND
    Non_Zero_Count = the number of nonzero elements in List(1 .. Index - 1)
    AND
    Product = product of the nonzero elements in List(1 .. Index - 1)
while Index <= Num_Values AND Non_Zero_Count < 10 loop
    if List(Index) /= 0 then
        Multiply List(Index) by Product
        Increment Non_Zero_Count
    end if
    Increment Index
end loop
```

This example is simple, but it's typical of the types of problems one gets into with loops. Specifying the initial values, terminating conditions, and the update of the condition inside the loop is not always easy. Should you initialize to 0 and increment before the other processing, or should you initialize to 1 and process and then increment? Should the terminating condition be a less-than comparison or a less-than-or-equal-to comparison? Using invariants to clarify your thinking about what the loop does helps you to write error-free loops.

Deskchecking, Walk-Throughs, and Inspections

Deskchecking

Tracing an execution of a design or program on paper.

When an individual programmer is designing and implementing a program, he or she can find many software errors with pencil and paper. **Deskchecking** the design solution is a very common method of manually verifying a program. The programmer writes down

essential data (variables, input values, parameters of subprograms, and so on) and walks through the design, marking changes in the data on the paper. Known trouble spots in the design or code should be double-checked. A checklist of typical errors (such as loops that do not terminate, variables that are used before they are initialized, and incorrect order of parameters on procedure calls) can be used to make the deskcheck more effective. A sample checklist for deskchecking an Ada program appears in Figure 2-2.

1. Does each object and module in the design have a clear purpose or function?
2. Can large modules be broken down into smaller pieces? (A common rule of thumb is that a program unit—package declaration, procedure, or function—should fit on one or two pages.)
3. Are all the assumptions valid? Are they well documented?
4. Are the preconditions and postconditions accurate assertions about what should be happening in the module they specify?
5. Is the design correct and complete, as measured against the program specification? Are there any missing cases? Is there faulty logic?
6. Has the design been clearly and correctly implemented in the programming language? Are features of the programming language used appropriately?
7. Is the program designed well for understandability and maintainability?
8. Are all parameter modes correct?
9. Are subprograms coded to be consistent with the interfaces shown in the design?
10. Are the actual parameters of calls to subprograms consistent with the parameters in the subprogram declarations?
11. Is each data object to be initialized set correctly at the proper time? Is each data object set before its value is used?
12. Do all loops terminate?
13. Is the design free of "magic" numbers? (A magic number is one whose meaning is not immediately evident to the reader.)
14. Has the programmer used any clever tricks that are difficult to understand?
15. Does each constant, type, and variable name have a meaningful name? Are comments included with the declarations to clarify the use of these data objects?

Figure 2-2
Checklist for Deskchecking an Ada Program

Have you ever been really stuck trying to debug a program, and showed it to a classmate or colleague who detected the bug right away? It is generally acknowledged that someone else can detect errors in a program better than the original author can. In an extension of deskchecking, two programmers can trade code listings and check each other's programs. Universities, however, often discourage students from examining each other's programs, for they perceive that this activity leads to cheating. Thus many students gain quite a bit of experience writing programs but don't have much opportunity to practice reading them.

Walk-Through

A verification method in which a *team* performs a manual simulation of the program or design.

Inspection

A verification method in which one member of a team reads the program or design line by line and the others point out errors.

Most sizable computer programs are developed by teams of programmers. Two extensions of deskchecking that programming teams use effectively are design/code **walk-throughs** and **inspections**. Both walk-throughs and inspections are formal team activities intended to move the responsibility for uncovering bugs from the individual programmer

to the group. Because testing is time consuming and errors cost more the later they are discovered, the goal of walk-throughs and inspections is to identify errors before testing begins—in fact, as early as possible in the software development process.

In a *walk-through* the team performs a manual simulation of the program with sample test inputs, keeping track of the program's data by hand on paper or a blackboard. Unlike thorough program testing, the walk-through is not intended to simulate all possible test cases. Instead, its purpose is to stimulate discussion of the way the programmer chose to design or implement the program's requirements.

At an *inspection*, a reader (not necessarily the program's author) goes through the design or code line by line. Inspection participants point out errors, which are recorded on an inspection report. Errors often are uncovered just by the process of reading aloud. Others may have been noted by team members during their preinspection preparation. As with the walk-through, the chief benefit of the team meeting is the discussion that takes place among team members. This interaction among programmers, testers, and other team members can uncover many program errors long before the testing stage begins.

If each participant has a copy of a numbered program listing, it is easier for the group to stay synchronized. In the inspection report, the errors are recorded by line number. Most Ada compilers have an option to generate a numbered program listing. Program `List`er, which we specified in Chapter 1, is another way to create this numbered program listing.

A walk-through or inspection is useful at a number of points in the software development process. At the high-level design stage, the design should be compared to the program requirements, to make sure that all required functions have been included and that this program or module correctly "interfaces" with other software in the system. At the low-level design stage, when the design has been filled out with more details, it should be reinspected before it is implemented. When the coding has been completed, a final inspection can be made of the compiled listings. This inspection (or walk-through) ensures that the implementation is consistent with both the requirements and the design. Successful completion of this inspection means that testing of the program can begin.

Program Testing

Eventually, after all the design verification, deskchecking, and inspections have been completed, it is time to execute the code *with the intention of finding any errors that may still remain*. That is, we start testing.

Many programmers are rather lax about program testing. Testing doesn't seem as interesting, challenging, or glamorous as writing the original program. Furthermore, after making a large investment in a particular solution, who wouldn't be reluctant to see it fail? Thorough testing, however, is an integral part of the programming process, and it can be as challenging as the software development.

The challenge is to try to "break" the code in as many different ways as possible. This goal requires you to change roles—to become the adversary, rather than the creator, of the program. For this reason it is often desirable to have someone other than the program's author perform the testing. In fact, software companies often have separate staffs for development and testing.

When you first began programming, you probably used the following testing methodology: Compile, then run the whole program over and over again on a set of data inputs provided by the instructor, debugging until your program successfully produced the appropriate outputs. This approach is somewhat effective with the small programs assigned to

beginners. Here is an example of the type of program that can be reliably tested on a single test run:

```
with Text_IO;
procedure Easy_Test is
begin
    Text_IO.Put_Line ("This is an easily tested program.");
end Easy_Test;
```

Program `Easy_Test` doesn't have any inputs, and there is only one possible output. No matter how many times you test this program, the result should be exactly the same.

Now that your programs are larger and more complex, however, such an approach is not very effective. Why? Simply because the sets of data inputs needed to establish program correctness are correspondingly larger and more complex. Many instructors provide their students with a single large data set to be used for validating a programming assignment. The output produced by running with this data is turned in along with a program listing, as evidence that the program was correctly implemented. Use of a common data set allows instructors to grade programming assignments consistently.

Unfortunately, however, many students take the existence of a class data set as evidence that the single-test-run approach that they used in their first course is sufficient for testing the more complex programs they write in later courses. We strongly suggest that the data set provided by the instructor be used only as a final test step, to verify a complete program after its various parts have been tested as independent units. What are independently testable units? Individual packages or single or related groups of subprograms can be tested more reliably than whole programs of any size. For this reason most of the discussion that follows focuses on testing strategies that verify pieces of programs.

The testing process is made up of a set of test cases that, taken together, allow us to assert that a program works correctly. We say "assert" rather than "prove" because testing does not generally provide a proof of program correctness.

The goal of each test case is to verify a particular program feature. For instance, we may design several test cases to demonstrate that the program correctly handles various classes of input errors. Other test cases may be designed to check the processing when a data structure (such as an array) is empty or when it contains the maximum number of elements.

Within each test case, a series of component tasks must be carried out.

- First we determine inputs that demonstrate the goal of this test case.
- Then we determine the *expected* behavior of the program for the given input. This task is often the most difficult one. For a math function, we might use a chart of values or a calculator to figure out the expected result. For a procedure with complex processing, we might use a deskcheck type of simulation or an alternative solution to the same problem.
- Next we run the program and observe the resulting behavior. This behavior may be observed by looking at either regular program output (something that happens on the terminal or is written on a printer) or output used just for testing.
- Finally we compare the expected behavior and the actual behavior of the program. If they are the same, the test case is successful. If not, an error has been found. In the latter case, we begin the activity known as debugging.

For now we are talking about test cases at a package, or procedure, level. It's much easier to test and debug packages of a program one at a time, rather than trying to get the whole program solution to work all at once. Testing at this level is called *unit testing*.

Let's apply these tasks to a test of a simple procedure called `Divider`, with the following specifications:

**Divider (Dividend: in Natural; Divisor: in Natural; Error: out Boolean;
Result: out Natural)**

Function: Execute the division of Dividend by Divisor.

Preconditions: None

Postconditions: **Error = (Divisor = 0)**

 Result = Dividend / Divisor (if not Error) or
 Result = undefined (if Error)

Here's the procedure we are going to test:

```
-- This procedure contains an error

procedure Divider (Dividend : in Natural;
                  Divisor  : in Natural;
                  Error    : out Boolean;
                  Result   : out Natural) is
-- Set Error to indicate if Divisor = zero.
-- If no error, set Result = Dividend / Divisor.
begin -- Divider
  -- Check for divisor of 0
  if Divisor = 0 then
    Error := True;                -- Just set Error; don't do division.
  else
    Result := Dividend / Divisor; -- Set Result from calculation.
  end if;
end Divider;
```

For simplicity, let's consider just two test cases: one that calls procedure `Divider` with a `Divisor` value of 0, and one that calls `Divider` with a nonzero value for `Divisor`. The procedure is called from a *test driver*, a program that sets up the parameter values and calls the procedure to be tested. (We talk more about test drivers later.) Here is a simple test driver. It is designed to execute both test cases. That is, it assigns the parameter values for Test 1, calls `Divider`, and prints the results; then it repeats the process with new test inputs for Test 2. We run the test and compare the values output from the test driver with the expected values.

```
with Text_IO;
with Integer_IO; -- Preinstantiated I/O Package
procedure Test_Driver is
  Answer : Natural;
  Error   : Boolean;

  -----
  -- Code for procedure Divider goes here --
  -----
```

```

procedure Put_Results (Dividend : in Natural;
                      Divisor   : in Natural;
                      Quotient  : in Natural;
                      Error     : in Boolean) is
begin
    Text_IO.Put ("Dividend =");
    Integer_IO.Put (Dividend);
    Text_IO.New_Line;
    Text_IO.Put ("Divisor =");
    Integer_IO.Put (Divisor);
    Text_IO.New_Line;
    Text_IO.Put ("Result =");
    if Error then
        Text_IO.Put_Line (" undefined");
    else
        Integer_IO.Put (Quotient);
        Text_IO.New_Line;
    end if;
end Put_Results;
-----
begin -- Test_Divider
    -- Test case #1
    Divider (Dividend => 8,      Divisor => 2,
              Result  => Answer, Error => Error);
    Put_Results (Dividend => 8,      Divisor => 2,
                 Quotient => Answer, Error => Error);
    Text_IO.New_Line (2);
    -- Test case #2
    Divider (Dividend => 8,      Divisor => 0,
              Result  => Answer, Error => Error);
    Put_Results (Dividend => 8,      Divisor => 0,
                 Quotient => Answer, Error => Error);
end Test_Divider;

```

In the first test case, we used values of 8 and 2 for `Dividend` and `Divisor`, respectively. The expected value for `Error` is `False`, and the expected value for `Result` is 4. But when we run our test, the actual value of `Error` is `True`! Our testing has uncovered an error, so we begin debugging. We quickly see that the value of `Error` is not set to `False` when the divisor is nonzero. Our solution is to add a statement in the `else` part of the procedure, setting `Error` to `False`. The resulting program looks like this:

```

-- This procedure contains an error.

procedure Divider (Dividend : in Natural;
                  Divisor   : in Natural;
                  Error     : out Boolean;
                  Result    : out Natural) is
-- Set Error to indicate if Divisor = zero.
-- If no error, set Result = Dividend / Divisor.
begin -- Divider

```

```

-- Check for divisor of 0.
if Divisor = 0 then
    Error := True;                -- Just set Error; don't do division.
else
    Error := False;
    Result := Dividend / Divisor; -- Set Result from calculation.
end if;
end Divider;

```

In the second test case, we picked values of 8 and 0 for `Dividend` and `Divisor`, respectively. The expected value for `Result` is undefined, and the expected value for `Error` is `True`. When we run this test our program is halted with a `CONSTRAINT_ERROR` exception. How can this be? The only action taking place in our procedure is the assignment of `True` to the parameter `Error`.

There is a subtle error in this procedure. This error is common when the value of an `out` parameter is undefined. In this test case the procedure has not assigned a value to `Result`—it is undefined. The value of this undefined formal parameter is copied back to the actual parameter, `Answer`. If this value is outside of the range permitted for `Answer`, `CONSTRAINT_ERROR` is raised. As `Answer` is type `Natural`, it's likely that the undefined value copied from `Result` was a negative number. Why didn't the undefined (negative) value in `Result` raise `CONSTRAINT_ERROR`? The range constraint of an object is checked only when *our* program assigns a value to it (through an assignment statement, declaration elaboration, or parameter passing). Ada does not check the initial value of an object. We can correct this program by assigning some nonnegative value to `Result`. Although we can assign any such value, we usually like to assign the largest or smallest possible value for the parameter. Here is our corrected procedure:

```

procedure Divider (Dividend : in Natural;
                  Divisor   : in Natural;
                  Error      : out Boolean;
                  Result     : out Natural) is
-- Set Error to indicate if Divisor zero.
-- If no error, set Result Dividend / Divisor.
begin -- Divider
    -- Check for divisor of 0.
    if Divisor = 0 then
        Error := True;                --Just set Error; don't do division.
        Result := Natural'Last;       --Set "undefined" Result.
    else
        Error := False;
        Result := Dividend / Divisor; -- Set Result from calculation.
    end if;
end Divider;

```

We were lucky to have discovered this error now. If the undefined value of `Result` had not been negative, `CONSTRAINT_ERROR` would not have been raised and we would have considered the test successful. In some future call to this procedure, the undefined value might be negative and result in a failure of our "tested" procedure. To prevent such errors in Ada programs you should:

Always assign a value to every out parameter, even if the specification states that the value is undefined.


```
-- Input an integer value and convert it to a Boolean value
-- as follows:  0 = False (no error). 1 = True (no error).
--             Any other value set the Error flag to True
--             and Boo to True.
.
.
.
```

Testing this procedure by running it with every possible data input isn't practical; the number of different integer values is clearly too large. In such cases we do not attempt exhaustive testing. Instead, we pick some other measurement as a testing goal.

You can attempt program testing in a haphazard way, entering data randomly until you cause the program to fail. Guessing doesn't hurt (except possibly by wasting your time and the computer's), but it may not help much either. This approach is likely to uncover *some* bugs in a program, but it probably will not find them all. There are, however, strategies for detecting errors in a systematic way.

One goal-oriented approach is to cover general classes of data. You should test at least one example of each category of inputs, as well as boundaries and other special cases. For instance, in procedure `Integer_To_Boolean` there are three basic classes of data: inputs of 0, 1, and "other." So you should plan three test cases, one for each of these classes. You could try more than three, of course. For example, you might want to try both negative and positive values in the "other" class. Testing, however, is time consuming, and a quick analysis of the program shows that different types of "other" data inputs are all treated the same way, so the additional test cases don't buy you much.

Black Box Testing

Testing a program or subprogram based on the possible input values, treating the code as a "black box."

There are other cases of data coverage. For example, if the input consists of commands, each command must be tested. If the input is a fixed-sized array containing a variable number of records, the maximum number of records should be tested; this maximum is the boundary condition. A way to test for robustness is to try one more than the maximum number of records. It is also a good idea to try an "empty" array or one that contains a single element. Testing based on data coverage is called **black box testing**. The tester must know the external interface to the module—its inputs and expected outputs—but does not need to consider what is being done inside the module (the inside of the black box). (See Figure 2-3.)

Code Coverage

Clear (Glass) Box Testing

Testing a program or subprogram based on covering all of the branches or paths in the code.

There are also a number of testing strategies based on the concept of *code coverage*: the execution of statements or groups of statements in the program. This testing approach is called **clear (or glass) box testing**. The tester must look inside the module (through the clear box) to see the code that is being tested.

For instance, one approach requires that every statement in the program be executed at least once. Another approach requires that the test cases cause every *branch*, or code section, in the program to be executed. (A simple if-then-else statement has two branches, one following `then` and the other following `else`.)

Path

A combination of branches that might be traveled when a program is executed.

Path Testing

A testing technique whereby the tester tries to execute all possible paths in a program or subprogram.

A similar type of code-coverage goal is to test program paths. A **path** is a combination of branches that might be traveled when the program is executed. In **path testing**, we try to execute all the possible program paths in different test cases.

The code-coverage approaches are analogous to the ways forest rangers might check out the trails through the woods before the hiking season opens. If the rangers wanted to test to make sure that all the trails were clearly marked and not blocked by fallen trees,

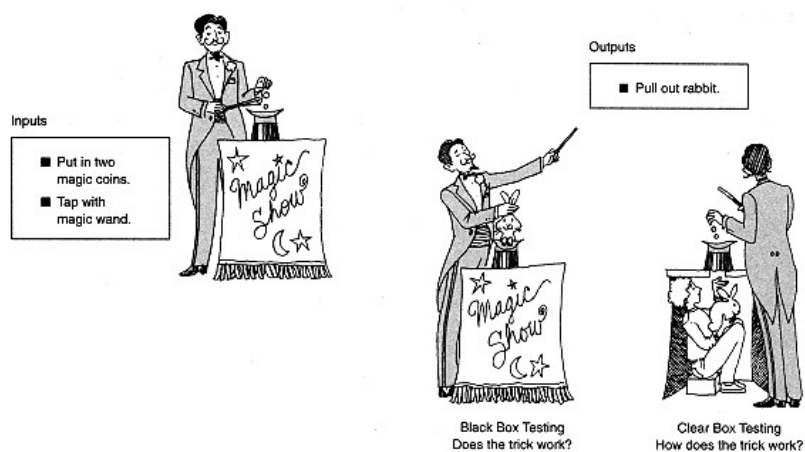


Figure 2-3
Testing Approaches

they would check each branch of the trails [see Figure 2-4(a)]. Alternatively, if they wanted to classify each of the various trails (which may be interwoven) according to its length and difficulty from start to finish, they would use path testing [see Figure 2-4(b)].

To create test cases based on code-coverage goals, we select inputs that drive the execution into the various program paths. How can we tell whether a branch or path is executed? One way to trace execution is to put debugging Put calls at the beginning of every branch, indicating that this particular branch was entered. Large programming projects often use software tools that help programmers track program execution automatically.

These strategies lend themselves to measurements of the testing process. We can count the number of paths in a program, for example, and keep track of how many have been covered in our test cases. The numbers provide statistics about the current status of testing; for instance, we could say that 75% of the branches of a program had been executed. When a single programmer is writing a single program, such numbers may be

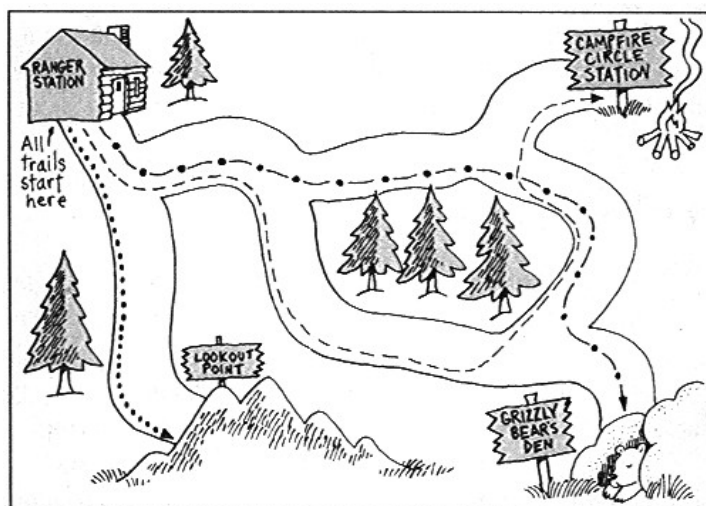


Figure 2-4a
Checking Out All the Branches

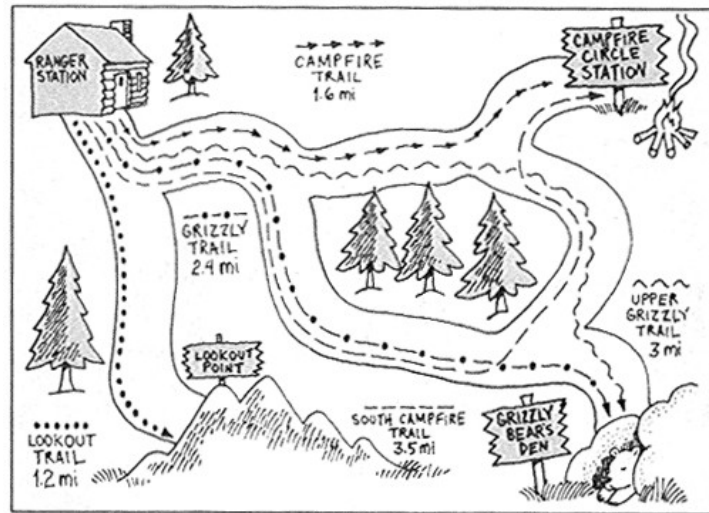


Figure 2-4b
Checking Out All the Trails

superfluous. In a software development environment with many programmers, however, statistics like these are very useful for tracking the progress of testing.

These measurements also can be used to indicate when a certain level of testing has been completed. Achieving 100% path coverage often is not a feasible goal. A software project might have a lower standard (say, 80% branch coverage) that the programmer who wrote the module is required to reach before turning the module over to the project's testing team. Testing in which goals are based on certain measurable factors is called *metric-based testing*.

Test Plans

Test Plan

A document showing the test cases planned for a program or module, their purposes, inputs, expected outputs, and criteria for success.

Deciding on the goal of the test approach—data coverage, code coverage, or (most often) a mixture of the two—precedes the development of a **test plan**. Some test plans are very informal—the goal and a list of test cases, written by hand on a piece of paper. Even this type of test plan may be more than you have ever been required to write for a class programming project. Other test plans (particularly those that are submitted to management or to a customer for approval) are very formal, containing the details of each test case in a standardized format. The main point of this discussion, however, can be summarized in a single word: *plan*.

For program testing to be effective, it must be planned. Determine the required or desired level of testing and plan the general strategy and test cases before testing begins. In fact, the planning for testing should begin before you write a single line of code.

Structured Integration Testing

Integration Testing

Testing that is performed on combined program modules that already have been independently tested.

Earlier we looked at a test driver to test procedure `Divider`, a program that supplied the test values, made the call, and printed out the results. In this section we discuss some concepts and tools that should help you put your test cases for individual modules together for structured testing of your whole program. The goal of this type of testing is to integrate the separately tested pieces, so it's called **integration testing**.

We can test a large, complicated program in a structured way through a method very similar to the structured approach used to design the program. The central idea is to divide

and conquer—to test pieces of the program independently and then to use the parts that have been verified as the basis for the next test.

Bottom-Up Testing

Test Driver

A program that sets up the testing environment by declaring and assigning initial values to variables, then calls the subprogram to be tested.

With this testing approach we verify the lowest level subprograms first, using **test driver** programs to call them. The test driver sets up data to provide inputs and calls the modules to be tested. (Remember the test driver for procedure `Divider`?)

A bottom-up approach can be useful in testing and debugging classes. Test drivers can also test "utility" subprograms, such as mathematical functions, independently of the programs that eventually call them. A bottom-up integration testing approach also can be effective in a group-programming environment, where each programmer writes and tests separate modules. The smaller, tested pieces of the program are verified later together in tests of the whole program.

As an example, let's assume that your assignment is to get together with two partners and write a fraction calculator program. Your team's design includes an object class for fractions. After working together on the package declaration for this class, the three of you decide to split the effort of coding and testing the program: One partner takes care of the fraction I/O (input/output) operations, another takes care of the command processing (done by the main program), while you write and debug the numeric operations (+, -, *, and /). Here is the portion of the fraction class package declaration relevant to your tasks:

```
package Fraction_Class is

    type Fraction is
        record
            Numerator    : Integer;
            Denominator   : Positive;
        end record;

    function "+" (Left : in Fraction; Right : in Fraction) return Fraction;

    function "-" (Left : in Fraction; Right : in Fraction) return Fraction;

    function "*" (Left : in Fraction; Right : in Fraction) return Fraction;

    function "/" (Left : in Fraction; Right : in Fraction) return Fraction;
    .
    .
    .
end Fraction_Class;
```

Your assignment isn't as trivial a task as it seems. You first must calculate the lowest common denominator, then adjust and add the numerators, and finally simplify the result.

You are obliged to code and test these low-level procedures bottom up, because your partners won't have the rest of the program finished by the time you are ready to test. You decide to write a simple test driver. The final completed program must provide prompts to ask the user for data and must check the input values for correctness. At this point, however, it is not important how the *real* data is input to the program; you just need to feed your functions data of the correct type as conveniently as possible. For instance, your test

data might be a line of input containing an operator (such as + or *) followed by four integers representing the numerators and denominators of two operands. Below is a very simple program that you can use to drive your tests. Each time the program is run, a single test case for each operation is executed.

```
with Fraction_Class;
use Fraction_Class;
with Integer_IO;      -- Preinstantiated I/O package
procedure Test_Fractions is
-- This program tests the fraction arithmetic in package Fraction_Class.
-- Input:  Operator character (+, -, *, or /) followed by four
--         integers representing the numerators and denominators of the
--         two fractions.

    type Operator_Type is ('+', '-', '*', '/');
    package Op_IO is new Text_IO Enumeration_IO (Enum => Operator_Type);

    Operator      : Operator_Type;
    Fraction_1    : Fraction_Class.Fraction;
    Fraction_2    : Fraction_Class.Fraction;
    Result        : Fraction_Class.Fraction;

begin
    Op_IO.Get (Operator);
    Integer_IO.Get (Fraction_1.Numerator);
    Integer_IO.Get (Fraction_1.Denominator);
    Integer_IO.Get (Fraction_2.Numerator);
    Integer_IO.Get (Fraction_2.Denominator);

    -- Echo inputs
    Integer_IO.Put (Item => Fraction_1.Numerator, Width => 1);
    Text_IO.Put ('/');
    Integer_IO.Put (Item => Fraction_1.Denominator, Width => 1);
    Text_IO.Put ("      ");
    Integer_IO.Put (Item => Fraction_2.Numerator, Width => 1);
    Text_IO.Put ('/');
    Integer_IO.Put (Item => Fraction_2.Denominator, Width => 1);
    Text_IO.New_Line;

    -- Calculate result
    case Operator is
        when '+' => Result := Fraction_1 + Fraction_2;
        when '-' => Result := Fraction_1 - Fraction_2;
        when '*' => Result := Fraction_1 * Fraction_2;
        when '/' => Result := Fraction_1 / Fraction_2;
    end case;

    -- Display result
    Integer_IO.Put (Item => Result.Numerator, Width => 1);
    Text_IO.Put ('/');
    Integer_IO.Put (Item => Result.Denominator, Width => 1);
    Text_IO.New_Line;
end Test_Fractions;
```

Note that the test driver gets the test data and calls the operations to be tested. It also provides written output to verify the inputs and outputs. This test driver does not do any error checking to make sure that the inputs are valid, however. For instance, it doesn't verify that denominators are positive values and it crashes with a `DATA_ERROR` if a character other than `+`, `-`, `*`, or `/` is entered for the operator. Remember that the goal of the test driver is to act as a skeleton of the real program, not to *be* the real program. Therefore, the test driver does not need to be as robust as the program it simulates.

You also could have the test driver process multiple test cases by adding a loop around the current processing.

Top-Down Testing

Stub

A special procedure or function that can be used in top-down testing to stand in for a lower level subprogram.

In a top-down approach, testing begins at the top levels, to see whether the overall logical design works and whether the interfaces between modules are correct. At each level of testing, this approach works on the assumption that the lower levels work correctly. We make this assumption by replacing the lower level subprograms with "placeholder" modules called **stubs**. A stub may consist of a single trace Put call indicating that we have reached the procedure, or of a group of debug Put calls showing the current values of the parameters. It also should assign values to `out` mode parameters.

As an example, consider the plight of your partner writing the main program for the fraction assignment. The design requires her to use the operations in package `Fractions_Class`, the very ones you and your other partner are writing. By using stubs, she can test her portion of the assignment before you complete your tasks. Here, for example, is a stub that could be used for the body of the function to add two fractions:

```
function "+" (Left : in Fraction; Right : in Fraction) return Fraction is
begin
    return (1,1); -- Simply return 1/1 no matter what the parameters.
end "+";
```

Combining the Approaches

Regression Testing

Re-execution of program tests after modifications have been made in order to ensure that the program still works correctly.

Once errors have been found and corrected, many of the earlier tests must be rerun to make sure that the corrections have not introduced any new problems. This rerunning of previous test cases is called **regression testing**.

While you are testing your part of the program with a test driver, your partners are writing and testing the rest independently, using other drivers and stubs to stand in for your modules. After you have tested your subprograms, you can add them to your partners' program, which includes the real routines that read the operands and write the results. Now when the program is run, you can verify that the I/O procedures actually input and set up the data correctly for your procedures. This level of testing verifies that the independently tested subprograms work together correctly; that is, it integrates the program pieces.

By now you probably are protesting that these testing approaches are a lot of trouble and that you barely have time to write your programs let alone "throwaway code" like stubs and drivers. Structured testing methods *do* require extra work. Test drivers and stubs are software items; they must be written and debugged themselves, even though you probably do not turn them in to a professor or deliver them to a customer. These programs are

part of a class of *software development tools* that take time to create but are invaluable in simplifying the testing effort.

These programs are like the scaffolding that a contractor erects around a building. It takes time and money to build the scaffolding, which is not part of the final product. But without it, the building can't be constructed. In a large program, where verification plays a major role in the software development process, creating these extra tools may be the only way to test the program.

Practical Considerations

It is obvious from this chapter that program verification techniques are time consuming and, in a job environment, expensive. It would take a long time to do all of the things discussed in this chapter, and a programmer has only so much time to work on any particular program. Certainly not every program is worthy of such cost and effort. How can you tell how much and what kind of verification effort is necessary?

A program's requirements may provide an indication of the level of verification needed. In the classroom your professor may specify the verification requirements as part of a programming assignment. For instance, you may be required to turn in a written test plan and to show what data inputs were used for each test case. Part of your grade may be determined by the completeness of your test data. In the work environment a customer often specifies the verification requirements in the contract for a particular programming job. For instance, a contract with a military customer may specify that formal reviews or inspections of the software product occur at various times during the development process.

Sections of a program that are particularly complicated or error-prone may need a higher level of verification effort. In these cases it is wise to start the verification process in the early stages of program development, in order to prevent costly errors in the design.

A program whose correct execution is critical to human life is obviously a candidate for a high level of verification. For instance, a program used for controlling the return of astronauts from a space mission would require a higher level of verification than would a program used for generating a grocery list. As a more down-to-earth example, consider the potential for disaster if a hospital's patient database system had a bug that caused it to lose information about patients' allergies to medications. However, a similar error in a database program that manages a Christmas card mailing list would have much less severe consequences.

Debugging with a Plan

Testing may indicate that our program contains errors. Debugging is the process of locating those errors. Once we find an error, correcting it may require us to change the program, the design, or even the specification.

Most Ada systems today have on-line debugging programs that make the debugging process much simpler. These programs usually allow you to execute one statement of your program at a time. This process is called *single stepping*. In addition, debugger programs usually let you set a *break point* at a particular statement in your program. Execution of a program with one or more break points halts just before executing a statement that contains a break point. When your program stops between single steps or at a break point, the debugger program allows you to examine the value of variables.

If your Ada system has a debugger facility, learn how to use it. Any tool that makes the task easier should be welcome. If you don't have a debugger, you can obtain the same results by inserting debugging Put calls in your Ada program to display the values of variables at various points in your program. The major disadvantage of this technique is that you must decide where to place the Put calls *before* you run the program. These debugging statements also can clutter up the program, making it harder to read.

We find it helpful to develop a top-down debugging plan for each error encountered. Basically, we make a hypothesis (a guess) as to what is causing the error. We then use the debugger or debugging Put calls to "prove" or "disprove" our hypothesis. Here is our algorithm for debugging with key words for each step shown in bold:

1. Write down the **symptoms** of the error.
2. Write down your **hypothesis** (best guess) as to what could cause this error. At first this might just be the name of a package or procedure that you think is not functioning correctly. Later it may be a control structure or single statement.
3. Write down a list of **variables to examine** whose values show whether your hypothesis is right or wrong. Also write down where in the program these variables should be examined and what values they should contain. This step is like the test plans we discussed earlier.
4. **Test your hypothesis** by setting break points (or insert debugging Put calls) at the locations determined in step 3. Run the debugger (or your program with debugging Put calls) and record the values of variables.
5. **Evaluate the evidence.**
 - a. If the evidence indicates that your hypothesis is correct, look for the error at the location you hypothesized. If you can't see the error, start again at step 2 with a more specific guess. Narrow your focus to smaller control structures (from packages to subprograms to control structures to individual statements).
 - b. If the evidence indicates that your hypothesis is not correct, you need to focus your attention to another part of the program. Return to step 2 and make another guess. If you can't make another guess, perhaps the error lies in your interpretation of the problem's specification or design. Review them and ask any questions you may have.

A warning about debugging: beware of the quick fix! Program bugs often travel in swarms, so when you find a bug, don't be too quick to fix it and run your program again. Often as not, fixing one bug generates another. A superficial guess about the cause of a program error usually does not produce a complete solution. In general, the time that it takes to consider all the ramifications of the changes you are making is time well spent.

If you constantly need to debug, there's a deficiency in your design process. The time that it takes to consider all the ramifications of the *design* you are making is time spent best of all.

Summary

Program verification is not something you attempt the night before your program is due. Design verification and program testing are activities that go on *throughout* the software life cycle.

We begin verification activities when the software specifications are developed. At this point, we formulate the overall testing approach and goals. Then as program design work begins, we apply these goals: We use formal verification techniques for parts of the program, hold design inspections, and give thought to test cases.

During the implementation phase, we develop the test cases and generate the test data to support them. Code inspections give the programmer extra support in debugging the program before it is ever run. When the code has been compiled and is ready to be run, we carry out unit (module-level) testing by writing and running simple test drivers. After these units have been completely tested, they are put together in integration tests. Once we have found and corrected errors, some of the previous tests need to be rerun to ensure that the corrections did not introduce new errors (regression testing).

Finally, we run acceptance tests of the whole system. When we have completed these tests, the software is put into use. Is the verification process finished? Hardly! More than half of the total life-cycle costs and effort generally come *after* the program becomes operational, in the maintenance phase. Some changes are made to correct errors in the original program; others are introduced to add new capabilities to the software system. In either case, we must do regression testing after any program modification.

Figure 2-5 shows how the various types of verification activities fit into the software development cycle. Throughout the life cycle one thing remains the same: The earlier in this cycle that we detect program errors, the easier (and less costly in time, effort, and money) they are to remove. Program verification is a serious subject; a program that doesn't work isn't worth the disk it's stored on.

<i>Analysis</i>	Make sure that specifications are completely understood. Understand testing requirements.
<i>Design</i>	Design for correctness (using preconditions and postconditions). Perform design inspections. Formally verify design (not discussed in this book). Plan testing approach.
<i>Code</i>	Understand programming language well. Perform code inspections. Add debugging assertion comments or debugging Put calls. Write test plan. Construct test drivers and/or stubs. Formally verify code (not discussed in this book).
<i>Test</i>	Unit test according to test plan. Debug as necessary using a debugger program when available. Integrate tested modules. Execute regression test after corrections.
<i>Delivery</i>	Execute acceptance tests of complete product.
<i>Maintenance</i>	Execute regression test whenever delivered product is changed to add new functionality or to correct detected problems.

Figure 2-5
Life-Cycle Verification Activities

Case Study: The Binary Search and Its Test Driver

This section illustrates how you can use a combination of techniques to develop and verify a procedure to implement the binary search algorithm. As indicated in this chapter, you can apply verification techniques throughout the software development process. In order to test the procedure, we develop a set of test data that is sufficient to verify the procedure, as well as a test driver that inputs this data and calls the search procedure.

We chose a search algorithm for this case study section because it's a non-trivial algorithm that you probably studied in your first course. Once you've designed, implemented, and verified this procedure, you can use it in other programs. Reusing pieces of software that have already been developed and tested is a very economical way to build programs. In order to use a procedure like this one in your own programs, however, you first must understand its interface—its inputs, outputs, and any assumptions about them. We document this interface with the following procedure specification:

Binary_Search (List: in; Value: in; Location: out)

Function: Searches List for Value; returns Location (an array index) if found, otherwise returns value of 0 in Location.

Input: List (an array to be searched)
Value (value to search for)

Preconditions: List's elements can be compared using = and < operators.
The values in List are in sorted order, from smallest to largest.
List's First is 1.

Output: Location

Postconditions: Value is in List and List(Location) = Value or
Value is not in List and Location = 0.

We now know what the procedure does, and that's all that the caller of the procedure needs to know. We do not know *how* the search is done, however. Although this information isn't needed by the caller, it is quite valuable to the programmer, so now let's talk a little about searching.

Searching

Given a sorted array of values and a particular value to search for, the first approach that comes to mind is the simplest: Compare `Value` to successive array elements, starting with the first element and searching until (1) we find `Value`, (2) we pass `Value`'s place in the sorted list without finding it, or (3) we have no more elements in the array to examine. The advantage of this algorithm is its simplicity; you probably have written a similar sequential search procedure. The disadvantage is the need to check every element in the array until `Value` is found. For instance, if `Value` is the last element in a 1000-element array, this searching algorithm needs to check all 1000 elements before it finds `Value`.

How can we describe this algorithm in terms of Big-O? In the worst case we have to check all the values in the list (N), which is $O(N)$. On the average, though, we have to check half the List ($1/2 N$). Since we ignore constants in determining the Big-O approximation, $1/2 N$ comparisons is also $O(N)$. So we can describe a sequential search as an $O(N)$ algorithm.

The Binary Search Algorithm

Can't we do better than that? Think of how you might go about finding a name in a phone book, and you can get an idea of a faster way to search. Let's look for the name "Dale." We open the phone book to the middle and see that the names there begin with M. M is larger than D, so we search the first half of the phone book, the section that contains A to M. We turn to the middle of the first half and see that the names there begin with G. G is larger than D, so we search the first half of this section, from A to G. We turn to the middle page of this section and find that the names there begin with C. C is smaller than D, so we search the second half of this section—that is, from D to G—and so on, until we are down to the single page that contains the name Dale. Figure 2-6 illustrates this algorithm.

How do we describe this algorithm using Big-O notation? To figure this out, let's see how many times we can split in half a list of N elements. Assuming that we don't find `Value` at one of the earlier midpoints, we have to divide `List` $\log_2 N$ times at the most, before we run out of elements to split. In case you aren't familiar with logs,

$$2^{\log_2 N} = N$$

That is, if $N = 1024$, $\log_2 N = 10$ ($2^{10} = 1024$). How does that apply to our searching algorithms? The sequential search is $O(N)$; in the worst case, we would have to search all 1024 elements of `List`. The binary search is $O(\log_2 N)$; in the worst case we would have to make $\log_2 N + 1$, or 11, search comparisons. A heuristic (a rule of thumb) tells us that a problem that is solved by successively splitting it in half is an $O(\log_2 N)$ algorithm. Figure 2-7 illustrates the relative growths of the sequential and binary searches measured in a number of comparisons. Now that we are convinced of how terrific it is, let's design the binary search procedure.

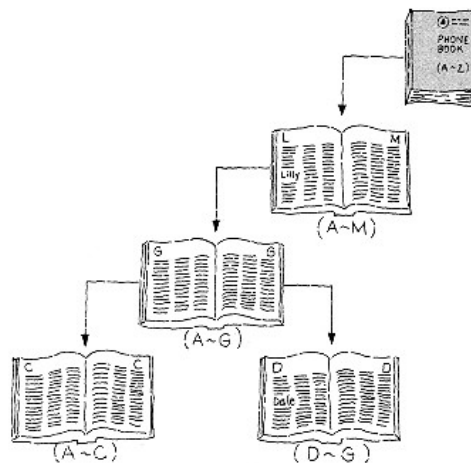


Figure 2-6
A Binary Search of the Phone Book

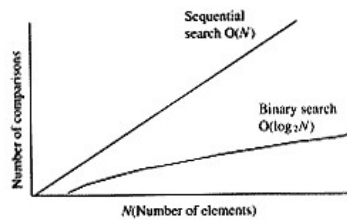


Figure 2-7
Comparison of Sequential and Binary Searches

Binary_Search

```

Current Search Area is List(1..List'Last)
Found := False
while more to search in Current Search Area and not Found loop
    Find Midpoint of Current Search Area
    if Value = List(Midpoint) then
        Found := True
    elsif Value < List(Midpoint) then
        Current Search Area becomes the first half of Current Search Area
    else
        Current Search Area becomes the second half of Current Search Area
    end if
end loop
if Found then
    Location := Midpoint
else
    Location := 0
end if

```

What do we mean by "Current Search Area"? We begin our search with the whole list to examine; that is, the current search area goes from `List(1)` to `List(List'Last)`. In each iteration, we split the current search area in half at `Midpoint`, and if `Value` is not found there, we search the appropriate half. The part of the list being searched at any time is the current search area. For instance, in the first iteration of the loop, if `Value` is less than the element at `Midpoint`, the new current search area goes from index 1 to the `Midpoint`. If `Value` is greater than the element at `Midpoint`, the new search area goes from the `Midpoint` to index `List'Last`. Either way, the current search area has been split in half. It looks like we can keep track of the boundaries of the current search area with a pair of indexes, `First_Index` and `Last_Index`. In each iteration of the loop, if `Value` is not found, one of these indexes resets to shrink the size of the current search area.

How do we know when to quit searching? There are two possible terminating conditions: (1) `Value` isn't in the list, or (2) `Value` has been found.

The first terminating condition, `Value` isn't in the list, occurs when there's no more to search in the current search area. This condition is a function of the values of `First_Index` and `Last_Index`, because these indexes define the current search area. If `First_Index < Last_Index`, there are still elements in the array to be examined. What if `First_Index = Last_Index`? `List(First_Index..Last_Index)` then would describe a range of array positions containing one element. As long as there is an element to examine, we continue the search. If `First_Index > Last_Index`, `List(First_Index..Last_Index)` is a null array: There are no

elements in it, so our search is over. Thus we know that the correct condition should be `while First_Index <= Last_Index`.

The second terminating condition occurs when `Value` has been found. This condition is expressed in the while loop as `while not Found`. That's simple enough. The whole condition is `while (First_Index <= Last_Index) and not Found`.

Does the loop terminate? Before entering the loop, we initialize `First_Index` to 1 and `Last_Index` to `List'Last`. In each iteration of the loop we reset either `First_Index` or `Last_Index` to redefine the current search area. Let's look again at this part of the algorithm.

```

elseif Value < List (Midpoint) then
    Current Search Area becomes the first half of Current Search Area
else
    Current Search Area becomes the second half of Current Search

```

If `Value` is less than the list element at `Midpoint`, we want to search the first half of the current search area. We can leave `First_Index` where it is, and move `Last_Index` to the `Midpoint`. Now we've cut the current search area in half. Similarly, if `Value` is greater than the `Midpoint` element, we can leave `Last_Index` where it is and move `First_Index` to the `Midpoint`.

```

elseif Value < List (Midpoint) then
    Last_Index := Midpoint
else
    First_Index := Midpoint

```

`First_Index` is getting bigger and `Last_Index` is getting smaller, so we know that they are heading toward each other. But do they ever pass each other? *Uh-oh!* What happens when `First_Index = Last_Index`? Now `Midpoint = First_Index = Last_Index`. We keep reassigning `First_Index` or `Last_Index` to the same value, and we find ourselves in an infinite loop!

At this point in the design we realize that, because we already know that `Value /= List (Midpoint)`, `Midpoint` shouldn't be in the new current search area. The correct design for updating the loop control variables is

```

elseif Value < List (Midpoint) then
    Last_Index := Midpoint - 1
else
    First_Index := Midpoint + 1

```

Now `First_Index` is always getting larger and `Last_Index` is always getting smaller; eventually they must pass each other, so we know the loop terminates. The design seems to be in pretty good shape, so we're ready to code the algorithm.

The Code—Version 1

-- This procedure contains an error.

```

procedure Binary_Search (List      : in Array_Type;
                        Value      : in Element_Type;
                        Location   : out Natural) is

```

```

-- Search List for Value;
-- Precondition:  List is sorted in ascending order.
-- Postcondition: If Value is found in List, Value = List(Location)
--                otherwise Location = 0.

Found      : Boolean; -- Has Value been found yet?
Midpoint   : Natural; -- Index of search area's midpoint
First_Index : Natural; -- First index in current search area
Last_Index  : Natural; -- Last index in current search area

begin -- Binary_Search
  -- Initialize
  Found      := False;
  First_Index := 1;
  Last_Index  := List'Last;

  -- Search until element found or the current search area is empty.
  -- Current search area is List (First_Index..Last_Index).

  while (First_Index <= Last_Index) and not Found loop
    -- Find middle element in the current search area.
    Midpoint := First_Index + Last_Index / 2;
    -- Compare Value to middle element in search area.
    if List(Midpoint) = Value then
      Found := True;
    elsif Value < List(Midpoint) then
      Last_Index := Midpoint - 1; -- Search area now 1st half
    else
      First_Index := Midpoint + 1; -- Search area now 2nd half
    end if;
  end loop;
  -- Set value of Location.
  if Found then
    Location := Midpoint;
  else
    Location := 0;
  end if;
end Binary_Search;

```

Developing a Test Driver

The next step is to run the program with the intention of detecting errors—that is, testing. But we cannot run a single procedure. Where is the program that calls this procedure? Where does the data to test it come from? We decide to write a simple test driver. Each execution of this program assigns values to `List` and `Value` and invokes the search routine. The input comes from the standard input file, which in this case is the tester at the keyboard.

Our test driver is *interactive*; it asks the tester to indicate the number of elements in the array to be searched. It then asks for the elements to be input. These values are stored in the array. Finally, the test driver asks the tester for values to search for. The test driver continues processing search values until the tester specifies zero (0) to indicate the test is

over. After each call to procedure `Binary_Search`, the test driver prints out the result of the call. Here is out test driver:

```
with Text_IO;
with Integer_IO;
procedure Test_Driver is
-----
-- Test driver for setting up array of values and
-- calling a search routine.

Max_Elements : constant := 10;
Quit_Value   : constant := 0;

subtype Element_Type is Integer;
type   Array_Type   is array(Positive range <>) of Element_Type;
subtype List_Type    is Array_Type (1..Max_Elements);

Num_Elements : Natural;      -- number of elements to search
List         : List_Type;    -- list of elements to search
Search_Value : Element_Type; -- value to search for in List
Location     : Natural;      -- index of Value in List

-----
-- Code for the binary search procedure goes here. --
-----

procedure Get_List (List : out Array_Type;
                   Count : out Natural) is
    -- Ask tester for list of values to put in search array.
    List_Size : Natural; -- Local copy of out parameter Count
begin -- Get_List
    Text_IO.Put_Line ("Enter number of test elements (0-10).");
    Integer_IO.Get (List_Size);
    -- Don't allow more than maximum elements.
    if List_Size > Max_Elements then
        List_Size := Max_Elements;
    end if;
    Count := List_Size; -- Copy value to out parameter.
    -- Prompt for test elements.
    Text_IO.Put_Line ("Enter the test elements in order, " &
                     "one per line.");
    -- Read in all the elements.
    for Index in 1..List_Size loop
        Integer_IO.Get (List(Index));
        Text_IO.Skip_Line;
    end loop;
end Get_List;

-----

procedure Print_List (List : in Array_Type) is
```

```

-- Print the elements in List.
begin -- Print_List
    Text_IO.New_Line;
    -- Print elements of List.
    Text_IO.Put ("Value: ");
    for Index in List'Range loop
        Integer_IO.Put (Item => List(Index), Width => 5);
    end loop;
    Text_IO.New_Line;
    -- Print index guide under elements.
    Text_IO.Put ("Index: ");
    for Index in List'Range loop
        Integer_IO.Put (Item => Index, Width => 5);
    end loop;
    Text_IO.New_Line(2);
end Print_List;

-----
begin -- Test driver
    -- Get list of array elements from tester.
    Get_List (List => List, Count => Num_Elements);
    -- Continue testing until tester wants to quit.
    loop
        -- Ask user for search value to test for.
        Text_IO.Put ("Input search value, or 0 to quit: ");
        Integer_IO.Get (Search_Value);
        exit when Search_Value = Quit_Value;
        -- Call the search procedure to be tested.
        Binary_Search (List      => List(1..Num_Elements),
                       Value     => Search_Value,
                       Location  => Location);
        -- Print the results after the search.
        Print_List (List(1..Num_Elements));
        if Location /= 0 then
            Text_IO.Put ("Search value found at location ");
            Integer_IO.Put (Item => Location, Width => 2);
        else
            Text_IO.Put ("Search value not found in list");
        end if;
        Text_IO.New_Line(2);
    end loop;
end Test_Driver;

```

Now that we have the driver, how do we test it? After all, it's a computer program and may have the same kinds of bugs that our "real" program has. One thing we can do is write a stub for procedure `Binary_Search` so that we can test the driver separately from the search routine. The stub might look like this:

```

procedure Binary_Search (List      : in Array_Type;
                        Value     : in Element_Type;
                        Location  : out Natural) is
-- Stub for procedure Binary_Search

```

```

begin
    Text_IO.Put_Line ("**** DEBUG ****   Entering Binary_Search");
    Text_IO.Put_Line ("                   Returning a value of 0");
    Location := 0;
end Binary_Search;

```

We compile the test driver program with this version of procedure `Binary_Search` and perform a few test runs. Note that we do not expect the answers to be correct, because we are not running the real search procedure. We just want to make sure that the mechanism for reading in the data, calling the search procedure, and printing the results works.

This test driver seems like a lot of work—its code is longer than the procedure it was designed to test. However, we can get some use out of it later. Note that just by changing the name of the search procedure it's called you can make this driver suitable for testing other search routines.

Developing a Test Plan

When we are satisfied with the test driver, we are ready to test procedure `Binary_Search`. First we replace the stub with the real procedure and recompile. Before running the program, however, we need to know what kind of inputs thoroughly test the procedure. Several types of tests come to mind: `List` may be empty (any search `Value`); `List` may contain only one element (check for `Value` in `List` and `Value` not in `List`); `List` may contain a number of elements (check for `Value` in first list position, `Value` in middle list position, `Value` in last list position, `Value` smaller than any list element, `Value` greater than any list element).

To make our tests easier to track, we document our planned inputs on a written test plan (see Figure 2–8). As we run the tests, we record the results.

Test 1

Goal: Test empty array.

Input: Num_Elements = 0, List = (), Value = any

Expected Result: Location = 0 (Value not found)

Actual Result: Location = _____

Test 2

Goal: Testing single element array.

Input: Num_Elements = 1, List = (4), Value = 4

Expected Result: Location = 1 (Value found)

Actual Result: Location = _____

Test 3

Goal: Testing single element array.

Input: Num_Elements = 1, List = (4), Value = 3

Expected Result: Location = 0 (Value not found)

Actual Result: Location = _____

Figure 2-8
Test Plan for `Binary_Search`

Test 4	
Goal:	Value in first list position.
Input:	Num_Elements = 8, Value = 1
	List = (1,3,5,7,9,11,13,15)
Expected Result:	Location = 1 (Value found)
Actual Result:	Location = _____
Test 5	
Goal:	Value in "middle" list position.
Input:	Num_Elements = 8, Value = 7
	List = (1,3,5,7,9,11,13,15)
Expected Result:	Location = 4 (Value found)
Actual Result:	Location = _____
Test 6	
Goal:	Value in last list position.
Input:	Num_Elements = 8, Value = 15
	List = (1,3,5,7,9,11,13,15)
Expected Result:	Location = 8 (Value found)
Actual Result:	Location = _____
Test 7	
Goal:	Value less than first list position.
Input:	Num_Elements = 8, Value = -1
	List = (1,3,5,7,9,11,13,15)
Expected Result:	Location = 0 (Value not found)
Actual Result:	Location = _____
Test 8	
Goal:	Value greater than last list position.
Input:	Num_Elements = 8, Value = 100
	List = (1,3,5,7,9,11,13,15)
Expected Result:	Location = 0 (Value not found)
Actual Result:	Location = _____
Test 9	
Goal:	Value falls between list values.
Input:	Num_Elements = 8, Value = 10
	List = (1,3,5,7,9,11,13,15)
Expected Result:	Location = 0 (Value not found)
Actual Result:	Location = _____

OK, let's run the program! Each different array size requires a separate execution of the program. But all the tests on a particular array (such as test cases 4 through 9) can be executed in one run of the program. Test cases 1 to 4 come out as expected. But test 5 goes into an infinite loop. Let's add some debugging Puts to our binary search procedure as shown in this version of the code:

The Code—Version 2

```
-- This procedure contains an error.

procedure Binary_Search (List      : in Array_Type;
                        Value     : in Element_Type;
                        Location  : out Natural) is
-- Search List for Value;
-- Precondition: List is sorted in ascending order.
-- Postcondition: If Value is found in List, Value = List(Location),
--               otherwise Location = 0.

    Found      : Boolean; -- Has Value been found yet?
    Midpoint    : Natural; -- Index of search area's midpoint
    First_Index : Natural; -- First index in current search area
    Last_Index  : Natural; -- Last index in current search area

begin -- Binary_Search
    Text_IO.Put_Line ("** DEBUG ** Entering Binary_Search "); -- **Debug**
    Text_IO.New_Line; -- **Debug**
    -- Initialize
    Found      := False;
    First_Index := 1;
    Last_Index  := List'Last;

-- Search until element found or the current search area is empty.
-- Current search area is List (First_Index..Last_Index).

    while (First_Index <= Last_Index) and not Found loop
        Text_IO.Put_Line ("** DEBUG ** Loop Entry "); -- **Debug**
        Text_IO.Put ("** DEBUG ** First_Index = "); -- **Debug**
        Integer_IO.Put (First_Index); Text_IO.New_Line; -- **Debug**
        Text_IO.Put ("** DEBUG ** Last_Index = "); -- **Debug**
        Integer_IO.Put (Last_Index); Text_IO.New_Line; -- **Debug**

        -- Find middle element in the current search area.
        Midpoint := First_Index + Last_Index / 2;
        Text_IO.Put ("** DEBUG ** Midpoint = "); -- **Debug**
        Integer_IO.Put (Midpoint); Text_IO.New_Line(2); -- **Debug**
        -- Compare Value to middle element in search area.
        if List(Midpoint) = Value then
            Found := True;
        elsif Value < List(Midpoint) then
            Last_Index := Midpoint - 1; -- Search area now 1st half
        else
            First_Index := Midpoint + 1; -- Search area now 2nd half
        end if;
    end loop;
-- Set value of Location.
    if Found then
        Location := Midpoint;
    else
        Location := 0;
    end if;
end if;
```

```

Text_IO.Put_Line ("** DEBUG ** Leaving Binary_Search "); -- **Debug**
Text_IO.Put ("** DEBUG ** Midpoint = "); -- **Debug**
Integer_IO.Put (Midpoint); Text_IO.New_Line; -- **Debug**

end Binary_Search;

```

When we run test case 5 with our second version, our debugging Puts produce a repeating series of values:

```

** DEBUG ** Loop Entry
** DEBUG ** First_Index =      4
** DEBUG ** Last_Index =      5
** DEBUG ** Midpoint =        6

```

Clearly Midpoint is not being correctly calculated. Yet somehow test cases 1 through 4 managed to produce correct output! Looking back at the code for procedure `Binary_Search`, we see that the calculation of Midpoint indeed is incorrect.

```
Midpoint := First_Index + Last_Index / 2;
```

Ada's precedence levels require that we use parentheses around the addition operation.

```
Midpoint := (First_Index + Last_Index) / 2;
```

Can you see why test cases 1 through 4 produced the correct output despite the error uncovered by test case 5? We make the correction and run the rest of our test plan. We also rerun the tests that previously passed to make sure our correction has not introduced another problem. All of our tests succeed.

Programmers often are surprised to find errors in their programs, especially when they have designed them carefully. However, even with a good design, stupid bugs often are introduced in the coding phase, as we have seen here.

Here is the final, corrected version of procedure `Binary_Search`, which you can use in your programs. We have removed the debugging lines because we no longer need them. Just in case, however, we have saved a version of the program with the debugging lines in it.

The Code—Final Version

```

procedure Binary_Search (List      : in Array_Type;
                        Value      : in Element_Type;
                        Location    : out Natural) is
-- Search List for Value;
-- Preconditions:  List is sorted in ascending order.
-- Postconditions: Value is in List and List(Location) = Value or
--                  Value is not in List and Location = 0.

    Found      : Boolean; -- Has Value been found yet?
    Midpoint    : Natural; -- Index of search area's midpoint
    First_Index : Natural; -- First index in current search area
    Last_Index  : Natural; -- Last index in current search area

```



```

begin -- Binary_Search
  -- Initialize
  Found      := False;
  First_Index := 1;
  Last_Index  := List'Last;

  -- Search until element found or the current search area is empty.
  -- Current search area is List (First_Index..Last_Index).

  while (First_Index <= Last_Index) and not Found loop
    -- Find middle element in the current search area.
    Midpoint := (First_Index + Last_Index) / 2;
    -- Compare Value to middle element in search area.
    if List(Midpoint) = Value then
      Found := True;
    elsif Value < List(Midpoint) then
      Last_Index := Midpoint - 1; -- Search area now 1st half
    else
      First_Index := Midpoint + 1; -- Search area now 2nd half
    end if;
  end loop;
  -- Set value of Location.
  if Found then
    Location := Midpoint;
  else
    Location := 0;
  end if;
end Binary_Search;

```

Exercises

1. True or False? (Correct the false statements.)

- (a) Software verification begins as soon as the program has been completely coded.
- (b) "Program verification" and "testing" are the same thing.
- (c) Testing is the only way to truly prove that a program is correct.
- (d) Compile-time errors are usually easier to detect and to correct than run-time errors.
- (e) The compiler can usually detect logical errors.

2. Have you ever written a programming assignment with an error in the specifications? If so, at what point did you catch the error? How damaging was the error to your design and code?

3. Explain why the cost of fixing an error is higher the later in the software cycle the error is detected.

4. Differentiate between a syntax error and a logical error in a program. When is each type of error likely to be found?

5. Indicate what kind of error (syntax or logical) is found in each of the following segments of code:

```

(a) -- Set In_Bounds flag according to value of Number.
    if Number <= Max then
        In_Bounds := True
    else
        In_Bounds := False;
    end if;

(b) -- Set In_Bounds flag and reset Number if it is out of bounds.
    if Number <= Max then
        In_Bounds := True;
    else
        In_Bounds := False;
    end if;
    Number := Max;

(c) type Error_Code is (OK, Out_Of_Bounds, Undefined, Wrong);
    .
    .
    .
    -- Process all error codes.
    Error := OK;
    loop
        exit when Error > Wrong;
        Process_Error (Error_Type => Error);
        Error := Error_Code'Succ (Error);
    end loop;

```

6. Explain how an expert understanding of your programming language can reduce the amount of time you spend debugging.
7. Give an example of a run-time error that might occur as the result of a programmer making too many assumptions.
8. Define "robustness." How can programmers make their programs more robust by taking a defensive approach?
9. What is meant by "designing for correctness"? Name two types of assertions described in this chapter that can contribute to a correct design.
10. Procedure `Average` takes `Data_List`, a constrained array of floats, and `Element_Count`, an integer that tells how many elements are stored in the array. It returns the float number `Data_Avg`, which is the average of the data stored in the array. Determine the preconditions and postconditions of this procedure.
11. Given the following preconditions and postconditions for procedure `Add_Element`, determine what would happen if we called `Add_Element` when the list was already full.

Preconditions: `Data_List` is not full.

Postconditions: `Data_List` = original `Data_List` plus new element added

12. (a) Define a "loop invariant."

(b) Give the loop invariant in the following segment of code:

```

Index := 1;
Found := False;
-- Loop Invariant : _____
while (Index <= Element_Count) AND NOT Found loop
    if Data_List(Index) = Target_Value then
        Found := True;
    end if;
end loop;

```

```

else
    Index := Index + 1;
end if;
end loop;

```

13. Give the loop invariants for the following segments of code:

```

(a) Quot := 0;
    -- Loop Invariant : _____
    while Number > 5 loop
        Number := Number - 5;
        Quot := Quot + 1
    end loop;

```

```

(b) Largest := Data(1);
    Index := 2;
    -- Loop Invariant : _____
    while Index <= 100 loop
        if Data(Index) > Largest then
            Largest := Data(Index);
        end if;
        Index := Index + 1;
    end loop;

```

14. Explain why the loop invariant contributes more to an understanding of the loop than does the condition on a while statement alone.

15. What should we always be sure to check when trying to verify the correctness of a loop design?

16. Fill in the blanks with True or False.

On entrance to an iteration of the loop body, the loop invariant must be _____ and the Boolean expression (in the while statement) must be _____. To exit from the loop, the loop invariant must be _____ and the Boolean expression (in the while statement) must be _____.

17. Write the loop that is described by the following loop invariant:

Loop Invariant: 1 <= Index <= Max_List + 1 AND
 0 is not found in List (1 .. Index - 1) AND
 Sum = Sum of List (1 .. Index - 1).

18. What is the purpose of conducting design and code inspections?

19. The following code fragment is intended to find the first array element with the value 0. It contains a logical error. Find and correct the error. (Hint: CONSTRAINT_ERROR is raised sometimes.)

```

Index := List'First;
loop
    exit when Index > List'Last or List (Index) = 0;
    Index := Index + 1;
end loop;

```

20. The following code fragment is intended to fill List with values from Data_File. The file contains one value per line.

```

Index := 0;
loop
    exit when Text_IO.End_Of_File (Data_File);
    Index := Index + 1;

```

```
Integer_IO.Get (File => Data_File, Item => List (Index);
end loop;
```

- (a) Identify a logical error that could occur due to an incorrect assumption about the input file.
- (b) Add one line to the code fragment so that the assumption you identified in part (a) need not be made.

21. The following code fragment prints out 37 "good" values and then a screen full of garbage. What's wrong? Correct the problem.

```
for Index in 1 .. Max_List_Size loop
    Integer_IO.Put (Item => List(Index));
end loop;
```

22. Is there any way a single programmer (for example, a student working alone on a programming assignment) can benefit from some of the ideas behind the inspection process?

23. When is it appropriate to start planning a program's testing?

- (a) during design or even earlier
- (b) while coding
- (c) as soon as the coding is complete

24. Differentiate between unit testing and integration testing.

25. Explain the advantages and disadvantages of the following debugging techniques:

- (a) inserting Put's that may be turned off with comment delimiters
- (b) using a Boolean flag to turn debugging Put's on or off
- (c) using a system debugger

26. Add some debugging Put calls to the following procedure to make it easier to test and debug. Don't worry about what the procedure does (nothing meaningful); just consider where debugging statements would help.

```
procedure Test_Me (First_Number : in Integer;
                  Second_Number : in Integer;
                  Answer : out Integer;
                  Out_Of_Bounds : out Boolean) is
    Temp : Integer;
    Term : Integer;
    Result : Integer;

begin
    Temp := First_Number * 2;
    Term := Calculate (Left => First_Number,
                     Right => Second_Number);
    Result := Term - Second_Number;
    Out_Of_Bounds := Result < 0 or Result > Temp;
    Answer := Result;
end Test_Me;
```

27. Differentiate between "data coverage" and "code coverage" in program testing. Is one better than the other?

28. *True or False?* We should try to test every possible input to a program in order to prove that it works correctly.

29. Describe a realistic goal-oriented approach to data-coverage testing of the procedure specified below:

```

procedure Find_Element (List   : in Integer_Array;
                       Target  : in Integer;
                       Index   : out Positive;
                       Found    : out Boolean);

-- Function:      Searches List for Target.

-- Preconditions:  The elements are in no particular order.
--                List may be empty.

-- Postconditions: Found = (Target was found in List)
--                Index = array index of the element if Found
--                Index = List'Last + 1 if not Found

```

30. A program is to read in a numeric score (0 to 100) and display an appropriate letter grade (A, B, C, D, or F).

- (a) What is the functional domain of this program?
- (b) Is exhaustive data coverage possible for this program?
- (c) Devise a test plan for this program.

31. Explain how paths and branches relate to code coverage in testing. Can we attempt 100% path coverage?

32. Differentiate between "top-down" and "bottom-up" integration testing.

33. You are writing a program that contains a loop controlled by a Boolean function `Terminate_Loop`.

```

loop
  exit when Terminate_Loop (Factor_1, Factor_2);

  -- Body of loop action goes here.

end loop;

```

You decide to work on testing the loop body, putting off testing the `Terminate_Loop` function until later. Write a stub for function `Terminate_Loop`.

34. Explain when a test driver is needed. If we advocate a top-down design approach, why would we ever need to test with a test driver?

35. Explain the phrase "life-cycle verification."

36. Explain why the preconditions in the following procedures are not needed:

```

(a) procedure Assert (Value : in Integer) is

    -- precondition: Value is a whole number.

(b) subtype Dozen is Integer range 1..12;
    .
    .
    .
    procedure Eggs (Chicken: in Dozen) is

    -- precondition: Chicken is greater than zero.

```

Programming Problems

1. You are to design, code, and test a *sequential search* procedure; then you are to compare this procedure to the binary search that was developed in the Case Study section of this chapter.

The Sequential Search

Given an array, List (sorted from smallest to largest) and a Value to search for, the sequential search uses the following approach: Compare Value to successive List elements, starting with the first element, and searching until (1) you find Value, (2) you pass Value's place in the sorted list without finding it, or (3) you have no more elements in the array to examine. You must develop the algorithm using a loop invariant.

When you finish coding the Sequential Search procedure, test it using the test driver from the Case Study section. (Simply replace the call to Binary Search with a call to Sequential Search.)

Comparing the Search Procedures

To compare the two searching procedures, you should modify each procedure to count the number of comparisons made before Value is found. Using the test driver, search for values in various positions in the array, and for values that are not in the array, using each search procedure. Following each call to a search procedure, you should print the search Value, location, and the comparison count. Print the results in a table format, with the following column headings:

```

Comparison Count

Value   Location   Binary Search   Sequential Search

```

Test Data

For more interesting results, you should increase the array size to 100, and fill the array. Remember that you must order the values from smallest to largest. Leave some gaps in the values in order to search for medium-sized values that are not in the array. Because of the large array size, you should modify Print_List to print the elements vertically, not horizontally. You only need to print the list once, before you begin to search for values. Modify the test driver to print all data to a file, in addition to the screen. At the end of your experiment, make a hard copy of the file to turn in.

Conclusions

Compare the lines of code and number of comparisons of the Sequential Search and Binary Search procedures. Are the results of your experiment consistent with the discussion in the Case Study section?

Turn In

1. the design of procedure Sequential_Search (It must contain a loop invariant.)
 2. listing of procedure Sequential_Search
 3. output from your comparison test runs
 4. your conclusions, in paragraph form
2. Your assignment is to design, code, and test a sorting procedure that uses the *insertion sort algorithm*. The procedure sorts the integer elements in an array that has the following declaration:

```
type Int_Array is Array (Positive range <>) of Integer;
```

The procedure has the following specifications:

Sort List (List : in out Int_Array)

Function: Sort the elements in List (List'Range) from largest to smallest value.

Preconditions: $1 \leq \text{List.Length}$

Postconditions: List (List'Range) are sorted in order of decreasing size.

The Insertion Sort Algorithm

The basic approach of the insertion sort is as follows: Each successive element in the list is inserted into its proper place relative to the other (already sorted) elements. We begin with the first element. There is only one element in the part of the list being examined, so it must be in its correct place. Now we put the second element in the list into its correct place, so that `List (1)` is larger than `List (2)`. (Note that they are not necessarily in their *final* positions in the sorted list; they are simply in their correct positions with respect to each other.) Then the value in the third array slot is put into its proper place, with respect to `List (1..3)`. This process continues until all the list elements have been sorted. The changing values of the arrays below illustrate the algorithm.

List.Length = 5

List	List	List	List	List
20	33	33	33	45
33	20	25	25	33
25	25	20	22	25
22	22	22	20	22
45	45	45	45	20

The sorting algorithm involves repeated examination of the elements in a list, which suggests the use of looping constructs. You should develop the design for procedure `Insertion_Sort` using loop invariants, then code the procedure. The code for the procedure must include Puts that output debugging data to a file called `DEBUG`.

Testing the Insertion Sort

Your next step is to create a test plan, showing the various test cases, along with inputs and expected outputs. The design of your test cases determines, in part, your grade.

To test the procedure, you should write a *batch* test driver. Your test driver should open the debug file, set up the array with values as specified in the test plan, and then call the `Insertion_Sort` procedure. The values in the array should be printed before and after sorting. The number of times that you need to set up array data and call `Insertion_Sort` depends on the test cases in your test plan. Make sure that the output of each test case clearly identifies it by test number. Write a stub for procedure `Insertion_Sort` to use in testing the test driver.

When your test driver is debugged, you are to comment out the `Insertion_Sort` stub, and replace it with the actual `Insertion_Sort` procedure. Record the results of your tests on the test plan. When you complete your testing, comment out the debugging Puts.

Turn In

1. the design of procedure `Insertion_Sort` (It must include loop invariants for any loops.)
2. the test plan, with results recorded
3. listing of the test driver and `Insertion_Sort` (Include the stub for `Insertion_Sort` and the debugging Puts in the listing, commented out.)
4. hard copy of the DEBUG file produced by your test

3—**Class Design and Implementation****GOALS**

After reading this chapter, you should be able to

- describe a class from three perspectives: application, abstraction, and implementation
- explain Ada's predefined types in terms of object classes
- understand and use Ada record discriminants
- specify the abstract view of a class as either an abstract data object (ADO) or abstract data type (ADT)
- use Ada's private types to encapsulate the internal details of a class

In Chapter 1 we defined a class as the description of a group of objects with similar properties and behaviors. Designing and implementing classes is the essence of object-oriented software development. In this chapter we strengthen your understanding of the class by analyzing several of Ada's predefined classes. With this background we then show you how you can design and implement your own classes.

Three Perspectives

In designing classes programmers wear many hats. We must determine the logical picture of the data in a particular program, choose the representation of the data, develop the operations, and encapsulate this entire arrangement. We consider classes from three different perspectives, or levels:

1. *Application level*: a way of modeling real-life objects in a specific context
2. *Abstract level*: an abstract collection of objects and its corresponding set of operations
3. *Implementation level*: a specific representation of the structure, its accessing operations, and its encapsulation in a programming language

Perhaps without even knowing it, you have used objects and classes in your earlier Ada programming experiences. Each of the predefined types that Ada provides is a class. As with all classes these types consolidate properties (data) and behavior (operations). You used these classes to declare objects whose properties can change (variables) and objects whose properties cannot change (named constants).

Scalar Types

Scalar Type

A data type in which each value is atomic (indivisible).

A **scalar type** is one in which each value is atomic; it cannot be decomposed. Ada's predefined types `Boolean`, `Character`, `Float`, `Integer`, `Natural`, and `Positive` are scalar classes. Type `String` is not a scalar class as a string can be decomposed into individual characters. While we restrict our discussion in this section to integer types, this discussion applies equally well to the other scalar classes.

You are certainly well versed in the *application* of integers. You have applied them to the solution of many problems. Keeping score at a sporting event, figuring out how many days you have left on an assignment, and checking the grade you received on a multiple choice exam all involve the use of integers.

At the *abstract level* we deal with the "what" questions. What is an integer? What operations can an integer perform? You learned this perspective on integers in your math classes and looked at it further in your first programming class. Figure 3-1 illustrates Ada's abstraction of the integer. Here the integer type is shown as a black box. We understand *what* this black box does but have no idea of what's inside it that knows *how* it is actually done.

Now let's consider the *implementation perspective* for integers. What is inside the box shown in Figure 3-1? You may think, as you should, that predefined scalar classes like `Integer` are very concrete; that there are no levels of abstraction below them. In reality, the engineers who designed your computer spent a good deal of time implementing integers.

Just how do we implement an integer? Integers are physically represented in different ways on different computers. They are stored as a group of binary digits (bits). The first implementation question that has to be answered is "how many bits should be used for each integer?" This decision limits the range of numbers that the computer can store. Then

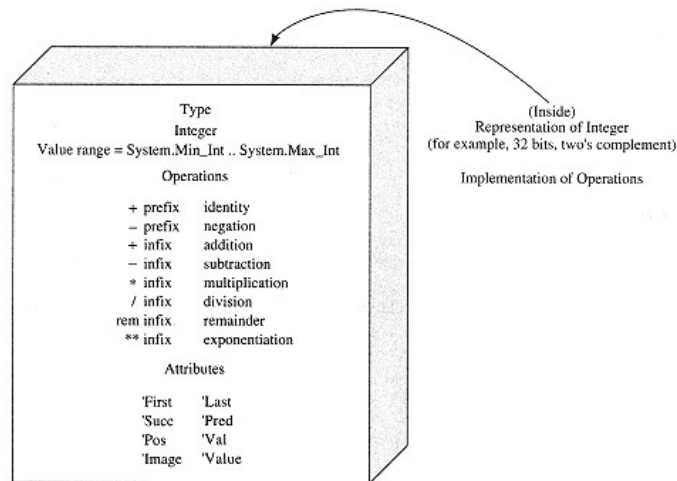


Figure 3-1
A Black Box Representing the Integer Class

the engineer has to select a method for encoding a decimal value on these bits. In the memory of one machine, an integer may be a binary-coded decimal. In a second it may be a sign-and-magnitude binary. In a third it may be represented in one's complement or two's complement notation. Although chances are you don't know what any of these terms mean, that hasn't stopped you from *using* integers. (You learn about these terms in a computer architecture course or an assembly language course, so we aren't going to explain them here.) Figure 3-2 shows the different decimal equivalents of a binary value.

Once a decision has been made on how to store integer values physically, the engineer still has to implement the necessary operations. The way that integers are physically represented determines how they are manipulated by electrical circuits in the computer.

The point of this discussion is that you have been dealing with an abstraction of integers since the very beginning. You have never had to deal with all the messy details of your computer's implementation of integers. The advantages of this *information hiding* are clear: You can think of the data and the operations in a logical sense and can consider their use without having to worry about implementation details. The lower levels are still there—they're just hidden from you. Ada encapsulates these details to prevent us from accessing them even if we wanted to.

Binary: 10011001

Decimal:	153	-25	-102	-103	99
	Unsigned	Sign and magnitude	Ones complement	Twos complement	Binary coded decimal

Figure 3-2
Some Possible Decimal Equivalents of an 8-Bit Binary Value

Composite Types

Composite Type

A data type that allows a collection of values to be associated with an object of that type.

A single integer can be very useful if we need a counter, a sum, or an index in a program, but generally we also must deal with objects that have lots of parts, such as a list. Such types are called *composite types*. Ada supplies two predefined composite types that contain multiple values: the record and the array. Let's look at each of these from our three perspectives. First we examine the abstract view of the structure—how we construct variables of that type and how we access individual components in our programs. Then from an application perspective, we discuss what kinds of things can be modeled using each structure. These two points of view are important to you as an Ada programmer. Finally, we look at how some of the structures may be implemented—how the "logical" accessing mechanism is turned into a location in memory. Understanding the implementation view of a predefined class is not critical for you as a programmer. You can just use predefined classes, as long as you know the syntax. As you read through the implementation sections and see the formulas that are needed to access an element of a composite class, you should appreciate why information hiding and encapsulation are necessary.

Records

The record is not available in all programming languages. FORTRAN, for instance, does not support records; however, COBOL, a business-oriented language, uses records extensively. In the C programming language, records are called structs.

Abstract Level

A *record* is a composite data type made up of a finite collection of not necessarily homogeneous elements called *fields*. Accessing is done directly through a set of named field *selectors*.

We illustrate the syntax and semantics of the accessing function within the context of the following declarations:

```
subtype Maker_String is String (1..10);
```

```
type Car_Rec is
  record
    Year   : Integer;
    Maker  : Maker_String;
    Price  : Float;
  end record;
```

```
My_Car : Car_Rec;
```

The record variable `My_Car` is made up of three components. The first, `Year`, is of type `Integer`. The second, `Maker`, is an array data type (`String` is a predefined array of characters). The third component, `Price`, is a `Float` number. The names of the components make up the set of selectors. A picture of `My_Car` appears in Figure 3-3.

The syntax of the accessing function is the record variable name, followed by a period, followed by the field selector for the component you are interested in.

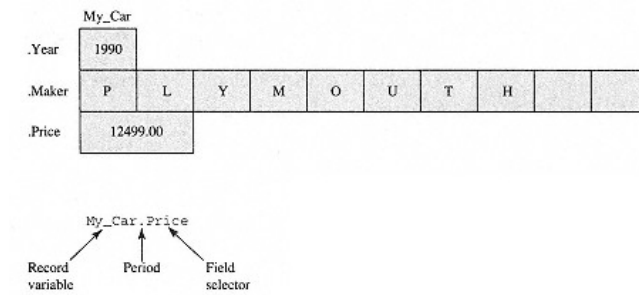
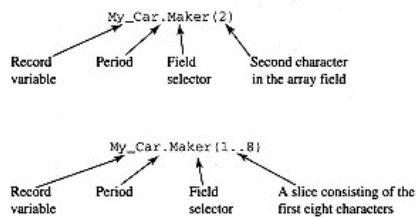


Figure 3-3
My_Car

If this expression is on the right-hand side of an assignment statement, a value is being extracted from that place (for example, `Price_Paid := My_Car.Price`). If it is on the left-hand side, a value is being stored in that field of the record (for example, `My_Car.Price := 12499.00`).

The `My_Car.Maker` field is an array whose elements are of type `Character` (Ada's predefined type `String`). You can access that array field as a whole (for example, `My_Car.Maker`), or you can access individual characters or a slice of characters by using an index or range with the field.



Application Level

Records are very useful for modeling objects that have a number of characteristics. The record data type allows us to collect various types of data about an object and to refer to the whole object by a single name. We also can refer to the different attributes or fields of the object by name. You probably have seen many examples of records used in this way to represent objects.

Records are also useful for defining other data structures, allowing programmers to combine information describing characteristics of the structure with the storage of the elements. One way to define a list is to declare the array that contains its elements and then to declare another variable to indicate how many of the slots in the array are being used.

```
Max_List : constant := 1000;  -- maximum elements in List
```

```
type Array_Type is array (1 .. Max_List) of Integer;
```

```
List      : Array_Type;      -- contains the elements
List_Count : Natural;        -- actual element count
```

If we want to call a procedure to do some processing on this list, we must pass both `List` and `List_Count`. (Only the elements in use should be processed, not all the slots in the array.)

Alternatively, we can combine the list elements and the other list information together in a record.

```

Max_List : constant := 1000;      -- maximum elements in List

type Array_Type is array (1 .. Max_List) of Integer;

type List_Type is
  record
    List_Count : Natural := 0; -- number of slots used
    Elements   : Array_Type;  -- contains the elements
  end record;

List : List_Type; -- contains everything needed to process the List

```

We make extensive use of records in this way when we develop representations of our own programmer-defined data structures.

This last example illustrates another feature of the record type. We can include an initial value for any field in the type declaration. Ada assigns this initial value to the field whenever a variable of the type is elaborated. For example, when the variable `List` is elaborated, Ada initializes `List.List_Count` to zero. When designing our own classes, we often use this feature instead of including an "initialize" operation to set the initial state of all objects in the class. We can eliminate the common error of forgetting to call an initialize operation for an object before using it.

Implementation Level

Implementation of any data type requires two things: (1) reservation of memory cells for the data and (2) functions to access the data. Before we go on to a concrete example, we digress for a moment and look at memory. The unit of memory assigned to hold a value is machine-dependent. Figure 3-4 shows several different memory configurations. In practice, how memory is configured is a consideration for the compiler writer. In order to be as general as possible, however, we use the generic term *cell*

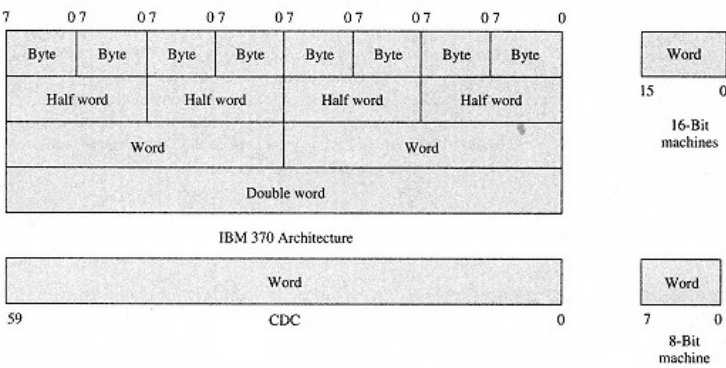


Figure 3-4
Memory Configurations

to represent a location in memory rather than "word" or "byte." In the examples that follow, we assume that an integer or character is stored in one cell and a float number in two cells.

The declaration statements tell the compiler how many cells are needed to represent the record. The name of the record then is associated with the characteristics of the record. These characteristics include

1. the location in memory of the first cell in the record, called the *base address* of the record
2. a table containing the number of memory locations needed for each element in the record

A record occupies a consecutive block of cells in memory. The record's accessing function calculates the location from a named field selector. The basic question is, Which cell (or cells) in this consecutive block do you want?

The base address of the record is the address of the first field in the record. To access any field, we need to know how much of the record to skip to get to the desired field. A reference to a record field causes Ada to examine the offset table to determine the field's offset from the beginning of the record. Ada then can generate the field's address by adding the offset to the base. Figure 3-5 shows such a table for `Car_Type`. If the base address of `My_Car` were 8500, the fields of this record would be at the following addresses:

Address of `My_Car.Year` = $8500 + 0 = 8500$
 Address of `My_Car.Maker` = $8500 + 1 = 8501$
 Address of `My_Car.Price` = $8500 + 11 = 8511$

Field	Length	Offset
Year	1	0
Maker	10	1
Price	2	11

Address	
8500	Year field (Length = 1)
8501	
8502	Maker field (Length = 10)
.	
.	
.	
8509	
8510	
8511	
8512	
	Price field (Length = 2)

Figure 3-5
`Car_Type`

Record Discriminants

The declaration of a record type may include one or more discriminants. (We restrict ourselves to records with a single discriminant.) A **discriminant** is

Record Discriminant

A special field in a record declaration that allows us to use the record declaration for different kinds or sizes of objects.

a special field in the record declaration that allows us to use that one declaration for different kinds of objects. Like any field in a record, the discriminant is declared with a type. The type of a discriminant is restricted; it must be a discrete type.

On page 84 we showed how a record may be used to encapsulate the length of a list and the array used to store the list's elements. Any list variable declared with this type holds up to 1000 elements. Using a discriminant, we can declare one record type that can be used to declare lists with different maximum sizes. Here is an example showing how we might do this:

```
type Array_Type is array (Positive range <>) of Integer;

type List_Type (Max_Size : Positive) is
  record
    List_Count : Natural := 0;           -- number of slots used
    Elements   : Array_Type (1..Max_Size); -- constrained array type
  end record;

Small_List : List_Type (Max_Size => 10);   -- 10 element maximum
Medium_List : List_Type (Max_Size => 100); -- 100 element maximum
Large_List  : List_Type (Max_Size => 1000); -- 1000 element maximum
```

The first declaration here is for an unconstrained array type—an array type for which the index range has not been specified. Only the type of the index and the type of the components are given for an unconstrained array type. The symbol <> is called "box." The identifier `Max_Size` given in parentheses after the record identifier `List_Type` is a discriminant. It is declared as `Positive`, a predefined discrete subtype. This discriminant is used in the declaration of field `Elements`. In particular, it is used as the upper bound in the range constraint supplied for the unconstrained type `Array_Type`.

Now look at the variable declarations for `Small_List`, `Medium_List`, and `Large_List`. Each of these declarations includes a value to be used for `Max_List`. The syntax for assigning a value to the discriminant is the same as associating formal and actual procedure parameters. We use named association here for extra clarity, but you also may use positional association. When these three variable declarations are elaborated, Ada allocates 10 elements for `Small_List`'s array, 100 elements for `Medium_List`'s array, and 1000 elements for `Large_List`'s array. We have used one record type to declare three different sizes of objects. Figure 3-6 illustrates these three variables. Notice that each record variable is a different size and that the discriminant, `Max_Size`, is stored as the first field in all of these records.

We could accomplish the same results by declaring three *different* record types, one with an array of 10 elements, one with an array of 100 elements, and one with an array of 1000 elements. The advantage to using the record with a discriminant is much more than the savings in typing declarations. We can use the record with a discriminant as a formal parameter. This allows us to pass any of the three list variables to the same procedure. Here's an example:

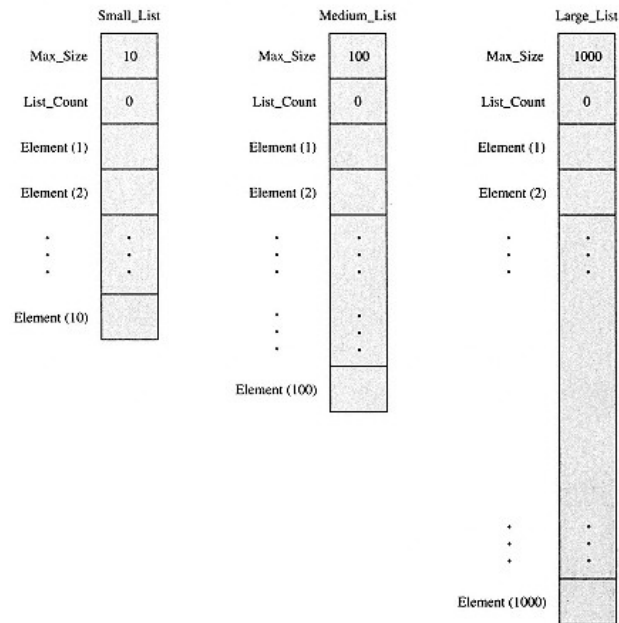


Figure 3-6
Three Variables of List_Type

```

procedure Put (List : in List_Type) is
begin
  for Index in 1..List.List_Count loop
    Integer_IO.Put (List.Elements(Index));
    Text_IO.New_Line;
  end loop;
end Put;
.
.
.
Put (List => Small_List);      -- Calls to procedure Put with
Put (List => Medium_List);     -- three different "kinds" of
Put (List => Large_List);      -- list records

```

If we declared three different list records rather than one record with a discriminant, we would have to write a separate `Put` procedure for each of the three different list types. We use records like this extensively for defining our own data structures.

One final note on discriminants. The value of a record discriminant may be accessed like any other component in the record. For example, the following statement displays the value 100:

```
Integer_IO.Put (Item => Medium_List.Max_Size, Width => 5);
```

However, once you have given the record discriminant a value in the declaration of a record object (variable or constant), you may not change it.* Thus the following statement is illegal:

```
Medium_List.Max_Size := 200; -- Illegal attempt to change a discriminant
```

One-Dimensional Arrays

Abstract Level

A one-dimensional array is a composite data type made up of a finite, fixed-size collection of ordered homogeneous elements. *Ordered* means that there is a first element, a second element, and so on. (It is the relative position of the elements that is ordered, not necessarily their values.) *Finite* indicates that there also is a last element. *Fixed size* means that the size of the array must be known at elaboration time; but it doesn't mean that all the slots in the array must contain meaningful values. Because the elements in an array must all be of the same type, they are physically *homogeneous*. In general, it is desirable for the array elements to be logically homogeneous, as well—that is, for all the elements to have the same purpose. (If we kept a list of numbers in an array of integers, with the length of the list—an integer—kept in the 0th array slot, the array elements would be physically, but not logically, homogeneous.)

The accessing mechanism of an array is *direct access*, which means we can access any element directly, without first accessing the preceding elements. The desired element is specified using an index, which gives its relative position in the collection. We talk later about how Ada uses the index and some characteristics of the array to figure out exactly where in memory to find the element. That's part of the implementation view, and the application programmer using an array doesn't need to be concerned with it.

What operations are defined for the array class? If Ada didn't have predefined arrays and we were defining arrays ourselves in the same manner as Ada, we would want to specify the following two operations (shown in an Ada-like pseudocode):

```
procedure Create_Array (An_Array      : out;
                       Index_Range   : in;
                       Element_Type  : in);
-- Procedure to create An_Array with the specified
-- Index_Range to contain data of type Element_Type

function Access_Element (An_Array : in;
                        Index      : in) return Memory_Location;
-- Function to return the location in memory
-- of the Index'th element of an array
```

Because arrays are predefined data types, however, the Ada programming language supplies a special way to perform each of these operations. Ada's syntax provides a type constructor for creating arrays in memory, and indexes as a way to directly access an element of an array.

*Ada has a mechanism for declaring and using unconstrained record types—records whose discriminants may be changed—that we have not discussed.

The construct operation is part of the declarations part of an Ada program. We declare a data type that defines what the array should look like and then use this type to declare an array variable. For example, a one-dimensional array can be declared with these statements:

```
-- Type declaration
type List_Array is array (1..10) of Integer;

-- Variable declaration
List : List_Array;
```

This declaration defines a linearly ordered collection of ten integer elements. Abstractly we can picture `List` as follows:

```

      List
1 First element
2 Second element
3 Third element
.
.
.
10 Last element
```

Each element in `List` can be accessed directly through its relative position in the array. The syntax of the accessing function is described by this EBNF definition:

```
indexed_component ::= name(expression)
```

The index expression may be as simple as a constant or a variable name, or as complex as a combination of variables, operators, and function calls. Whatever the form of the expression, it must result in a value that is compatible with the index type.

The indexing expression must have a value in the range declared for the array; in the case of `List`, the value must be between 1 and 10. The semantics (meaning) of the accessing function is "Locate the element associated with the indexing expression in the collection of elements identified by array-name."

The accessing function can be used in two ways:

1. to specify a place into which a value is to be copied such as

```
List(2) := 5;
```

2. to specify a place from which a value is to be extracted such as

```
Number := List(Index);
```

Declaring an array and accessing individual array elements are operations predefined in nearly all high-level programming languages. Ada has several additional operations. It allows us to access groups of consecutive elements in an array. These groups of elements are called *slices*. A slice is specified by giving a range rather than a single subscript. For example, the expression

```
List(2..5)
```

specifies an array of four elements. We may use a slice of an array just as we would an entire array.

Other operations predefined by Ada for arrays include catenation (&), relational operations (= and /= for all component types; <, <=, >, and >= for arrays with discrete components), logical operations (for arrays with Boolean components), and attributes (for example, 'First, 'Length, and 'Range). Finally, Ada's array aggregates provide a convenient way to assign values to an array object as illustrated by the following assignment statements:

```
List := (57, 3, 212, 16, 92, 43, 82, 423, 21, 101);
```

```
List := (1..10 => 0);
```

Application Level

A one-dimensional array is the natural structure for the storage of lists of like data elements. However, the operations available for an array object are so limited (only creation and direct access) that the *major* application of arrays is the implementation of higher level classes. You will find arrays encapsulated in the implementation of many of the classes described in this book.

Implementation Level

Of course when you use an array in an Ada program you do not have to be concerned with all of the implementation details. You have been dealing with an abstraction of the array since the very beginning. You never have to deal with all the messy details of your system's implementation of arrays that we describe in this section.

An array declaration statement tells the compiler how many cells are needed to represent that array. The name of the array then is associated with the characteristics of the array. These characteristics include

1. the upper bound of the index range (Upper)
2. the lower bound of the index range (Lower)
3. the location in memory of the first cell in the array, called the *base address* of the array (Base)
4. the number of memory locations needed for each element in the array (Size)

The information about the array characteristics is stored in a table called an *array descriptor*. When Ada comes across references to an array element, it uses this information to execute code that calculates the element's location in memory at run time.

How are the array characteristics used to calculate the number of cells needed and to develop the accessing functions for the following arrays? As before, we assume that an integer or character is stored in one cell and a floating point number in two cells.

```
type Data_Array   is array (1..10)    of Integer;
type Value_Array  is array (-3..2)    of Float;
type Letter_Array is array ('A'..'Z') of Integer;
```

```
Data           : Data_Array;
Values         : Value_Array;
Letter_Count   : Letter_Array;
```

These arrays have the following characteristics:

	<i>Data</i>	<i>Values</i>	<i>Letter_Count</i>
Upper	10	2	'Z'
Lower	1	-3	'A'
Base	unknown	unknown	unknown
Size	1	2	1

We can look at the declaration for *Data* and immediately see that ten cells are required. The index type of *Values* is more complicated, however. To determine the number of cells needed by a particular index type, use take the ordinal position (*'Pos*) of the upper bound of the index type, subtract the ordinal position of the lower bound of the index type, then add 1. The result is multiplied by the number of cells per element (*Size*) to get the number of cells needed for the whole array. The formulas for arrays with an *Integer* index like *Data* and *Values* and for arrays with a *Character* index like *Letter_Count* are

Number of cells = (*Integer'Pos*(Upper) - *Integer'Pos*(Lower) + 1) * *Size*

Number of cells = (*Character'Pos*(Upper) - *Character'Pos*(Lower) + 1) * *Size*

The following table applies this formula to the arrays above:

<i>Index Type</i>	<i>'Pos</i> (Upper)	-	<i>'Pos</i> (Lower)	+	1	=	<i>Number of Elements</i>	*	<i>Size</i>	=	<i>Number of Cells</i>
(1..10)	10	-	1	+	1	=	10	*	1	=	10
(-3..2)	2	-	(-3)	+	1	=	6	*	2	=	12
('A'..'Z')	90	-	65	+	1	=	26	*	1	=	26

The Ada compiler usually assigns memory cells to variables in sequential order. If, when the three declarations above are encountered, the next memory cell available to be assigned is, say, 100, then the memory assignments are as follows. (We have used 100 to make the arithmetic easier.)

	<i>Data</i>	<i>Address</i>		<i>Values</i>	<i>Address</i>		<i>Lettercount</i>	<i>Address</i>
(1)		100	(-3)		110	('A')		122
(2)		101	(-2)		112	('B')		123
.	.		.	.		.	.	
.	.		.	.		.	.	
.	.		.	.		.	.	
(10)		109	(2)		120	('Z')		147

Now we have determined the base address of each array: `Data` is 100, `Values` is 110, and `Letter_Count` is 122. The arrangement of these arrays in memory gives us the following relationships:

Given	The program must access
	
<code>Data(1)</code>	100
<code>Data(9)</code>	108
<code>LetterCount('A')</code>	122
<code>LetterCount('C')</code>	124
<code>Values(-1)</code>	114
<code>Values(0)</code>	116

The accessing function that gives us the position of an element in a one-dimensional array associated with the expression `Index` is

$\text{Address}(\text{Index}) = \text{Base} + \text{Offset of the Index}^{\text{th}} \text{ element}$

How do we calculate the offset? The general formula is

$\text{Offset} = ('Pos(\text{Index}) - 'Pos(\text{Lower})) * \text{Size}$

The whole accessing function becomes

$\text{Address}(\text{Index}) = \text{Base} + ('Pos(\text{Index}) - 'Pos(\text{Lower})) * \text{Size}$

Let's apply this formula and see if we do get what we have shown we should.

	<i>Base</i>	+	(<i>'Pos(Index)</i>	-	<i>'Pos(Lower)</i>	*	<i>Size</i>)	=	<i>Address</i>
<code>Data(1)</code>	100	+	(1	-	1)	*	1	=	100
<code>Data(9)</code>	100	+	(9	-	1)	*	1	=	108
<code>Letter_Count('A')</code>	122	+	(65	-	65)	*	1	=	122
<code>Letter_Count('C')</code>	122	+	(67	-	65)	*	1	=	124
<code>Values(-1)</code>	110	+	((-1)	-	(-3))	*	2	=	114
<code>Values(0)</code>	110	+	(0	-	(-3))	*	2	=	116

Ada stores an array descriptor with every array object. The `Base` entry of the descriptor is filled in when the array variable is elaborated. The values of `Upper` and `Lower` are unknown when the array type is unconstrained. These entries are filled in when an array object is constrained during its elaboration.

The array descriptor is used during compilation and run time for many different purposes, including error checking and the creation of array slices. For example, after an indexing expression has been calculated, it is compared to the information stored in the array descriptor. If the calculated index is outside the array's index range, `CONSTRAINT_ERROR` is raised.

As we mentioned at the beginning of this section, when you use an array in an Ada program you do not have to be concerned with all of these implementation details. The advantages of doing so are very clear: You can think of the data and the operations in a logical sense and can consider their use without having to worry about implementation details. The lower levels are still there—they're just hidden from you. We strive for this same sort of separation of the abstract and implementation views in the programmer defined classes discussed in the remainder of this book.

Two-Dimensional Arrays

Abstract Level

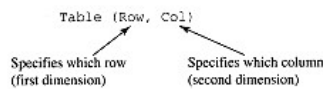
Most of what we have said about the abstract view of a one-dimensional array applies as well to a two-dimensional (or multidimensional) array. It is a composite data type made up of a finite, fixed-size collection of ordered homogeneous elements. Its accessing mechanism is direct access: A pair of indexes specifies the desired element by giving its relative position in the collection. Slices are *not* available for two-dimensional arrays.

A two-dimensional array is a natural way to represent data that is logically viewed as a table with columns and rows. The following example illustrates the syntax for the type constructor:

```
type Table_Array is array (1 .. 10, 1 .. 5) of Integer;
```

```
Table : Table_Array;
```

The abstract picture of this structure is a grid with rows and columns as illustrated in Figure 3-7(a). The accessing function for the two-dimensional array is as follows:

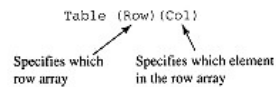


In Ada, we also can define an array of arrays. The declaration of an array type includes an index type (or types) and a component type. Although an index type must be a discrete type, the component of an array may be any type. Thus, a component type may even be another array type. Here is an example of an array of arrays:

```
type Row_Array is array (1..5) of Integer;
type Table_Array is array (1..10) of Row_Array;
```

```
Table : Table_Array;
```

The abstract picture of this structure is illustrated in Figure 3-7(b). As with two-dimensional arrays, we use two indexes to access an element in an array of arrays. We must, however, enclose each index in its own set of parentheses.



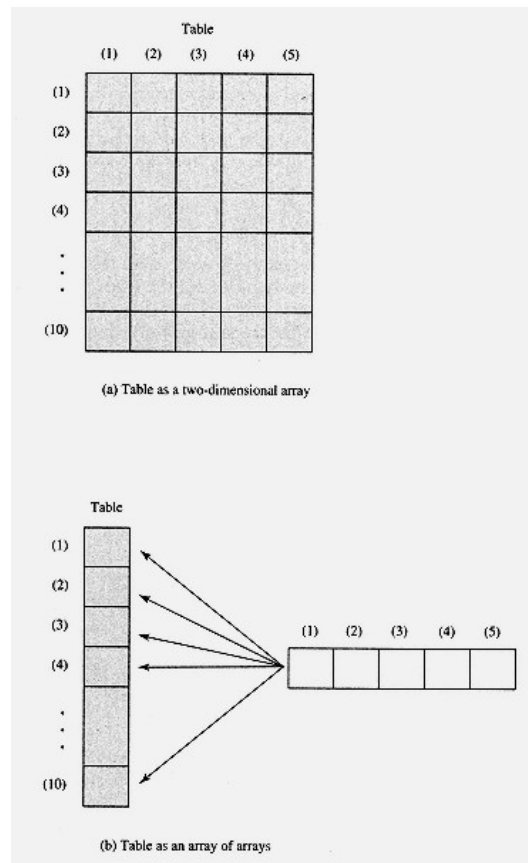


Figure 3-7
Two Implementations of Table

Arrays of arrays give us more flexibility than two-dimensional arrays. Because an array of arrays involves only one-dimensional arrays, we can use slices to access a portion of a single row or a number of complete rows. The following examples illustrate accessing elements and slices:

<i>Expression</i>	<i>Explanation</i>
Table	Entire array
Table(2)	The second row
Table (7..10)	The last four rows
Table (7)(2)	The second element in the seventh row
Table (7)(2..4)	A slice of the seventh row
Table (7..10)(2..4)	ILLEGAL

We often use expressions like these to pass a portion of our data to a subprogram.

Application Level

As mentioned in the previous section, a two-dimensional array is the ideal data structure for modeling data that is logically structured as a table with rows and columns. The first dimension represents rows, and the second dimension represents columns. Each element in the array contains a value, and each dimension represents a relationship. As with the one-dimensional array, the operations available for a two-dimensional array object are so limited (only creation and direct access) that the major application is the implementation of higher level classes.

Implementation Level

The implementation of two-dimensional arrays involves the mapping of two indexes to a particular memory cell. The mapping functions are more complicated than those for one-dimensional arrays. We do not give them here. You will learn them in an assembly language course.

Data Structures

A data structure is a class composed of elements of other classes whose organization is characterized by accessing operations that are used to store and retrieve the individual elements. Although the predefined record and array classes fit this definition, we usually restrict the term *data structure* to classes designed and implemented by programmers. Some people now use the more descriptive term *collection class* in place of data structure.

Data Structure (Collection Class)

A class composed of elements of other classes whose organization is characterized by accessing operations that are used to store and retrieve the individual elements.

A **data structure** is defined by (1) the logical arrangement of data elements (its structure), combined with (2) the set of operations we need to access the elements.

Let's look at a real-life example: a library. A library can be decomposed into its component elements—books. The collection of individual books can be arranged in a number of ways, as shown in Figure 3-8. Obviously the way the books are physically arranged on the shelves determines how one would go about looking for a specific volume. The particular library we're concerned with uses closed stacks; it doesn't let its patrons get their own books. If you want a book, you must give your request to the librarian, who gets the book for you.

The library "data structure" is composed of elements (books) in a particular physical arrangement; for instance, it might be ordered on the basis of the Dewey decimal system or the Library of Congress system. Accessing a particular book requires knowledge of the arrangement of the books. The library user doesn't have to know about the structure; users access books only through the librarian. The structure of the library is encapsulated.

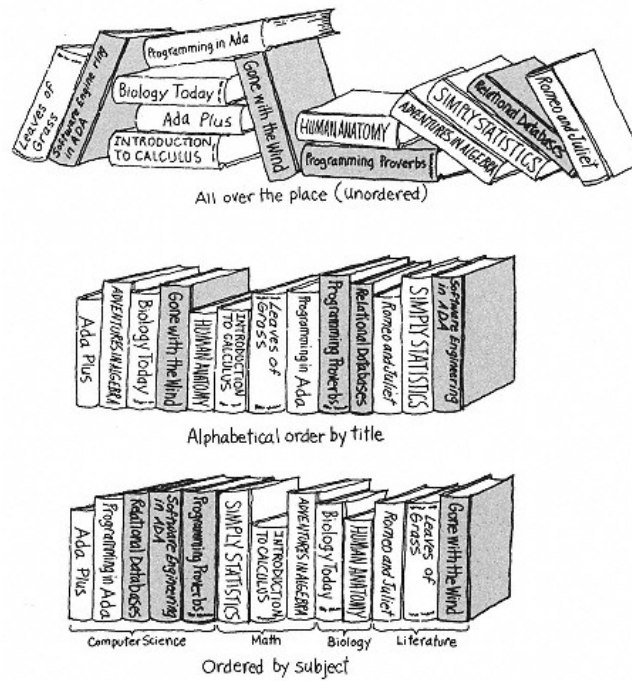


Figure 3-8
A Collection of Books Ordered in Different Ways

The physical structure and abstract picture of the books in the library are not the same. The books might be arranged on the shelves according to the Library of Congress system. The card catalog provides abstract views of the library—ordered by subject, author, or title—that are different than its physical arrangement.

Let's look at our library analogy from our three viewpoints. At the *application level*, there are entities (objects) like the Library of Congress, the Dimsdale Collection of Rare Books, and the Plattsburgh City Library.

At the *abstract level*, we deal with the "what" questions: What is a library? What services (operations) can a library perform? The library may be seen abstractly as "a collection of books," for which these operations are specified:

- Check out a book.
- Check in a book.
- Reserve a book that is currently checked out.
- Pay a fine for an overdue book.
- Pay for a lost book.

How the books are organized on the shelves is not important at the abstract level, because the patrons don't actually have access to the books. The abstract viewer of library services is not concerned with how the librarian actually organizes the books in the library. The library user only needs to know what is the correct way to invoke the desired operation. For instance, here is the user's view of the operation to check in a book: Present book at check-in window of library from which book was checked out, and receive fine slip if the book is overdue.

At the *implementation level*, we deal with the answers to the "how" questions: How are the books cataloged? How are they organized on the shelf? How does the librarian process a book when it is checked in? For instance, the implementation information includes the fact that the books are cataloged according to the Library of Congress system and arranged in four levels of stacks, with 14 rows of shelves on each level. The librarian needs such knowledge to be able to locate a book. This information also includes the details of what happens when each of the operations takes place. When a book is checked back in, the librarian may use the following algorithm to implement the check-in operation:

Check_In_Book

Examine due date to see whether the book is late.

```
if book is late then
    Calculate fines.
    Issue fine slip.
end if
```

Update library records to show that the book has been returned.

Check reserve list to see if someone is waiting for the book.

```
if book is on reserve list then
    Put the book on the reserve shelf.
else
    Replace the book on the proper shelf, according to the library's shelf arrangement
    scheme.
end if
```

All this, of course, is invisible to the library user.

The goal of our design approach is to hide the implementation level of a data structure from the programmer who uses it, just like Ada hides the implementation of the integer and array from us. Picture a wall separating the application level from the implementation level, as shown in Figure 3-9. Imagine yourself on one side and another programmer on the other. How do the two of you, with your separate views of the data, communicate across this wall? Similarly, how do the library user's view and the librarian's view of the library come together? The library user and the librarian communicate through the abstract view which provides the specification of the accessing operations without telling how the operations work. It tells *what* but not *how*. For instance, the abstract view of checking in a book can be summarized in the following specification:

Check_In_Book (Library : in out, Book : in; Fine_Slip : out)

Function: **Check in a book.**

Input: **Book, Library**

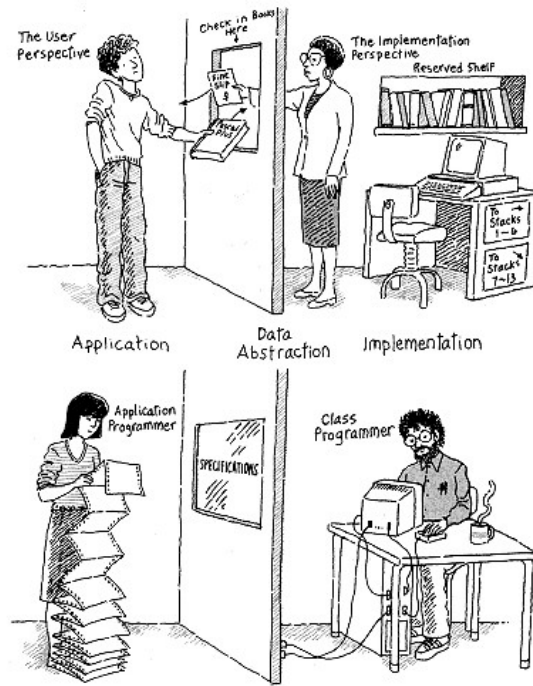


Figure 3-9
Communication Between the Application Level and Implementation Level

Preconditions: Book was checked out of this Library.

Output: Fine_Slip, Library (changed)

Postconditions: Book is overdue and Fine_Slip is issued or
 Book is not overdue and no Fine_Slip is issued

Library = original Library + Book

The only communication from the user into the implementation level is in terms of input specifications and allowable assumptions—the preconditions of the accessing routines. The only output from the implementation level back to the user is the transformed objects described by the output specifications, or postconditions, of the routines. The abstract view hides the element organization and algorithms but provides windows through the specified accessing operations.

When you write a program as a class assignment, you often deal with data at all three levels. In a job situation, however, you may not. Sometimes you may program an application that uses a class that has been implemented by another programmer. Other times you may develop classes that are used by other programmers. In this book we ask you to move back and forth between these views.

Encapsulation

An important feature of a data structure is that the arrangement of the elements and the way they are accessed are encapsulated by the programmer. In Ada, we use the package to encapsulate data structures. We illustrate how this is done with an example.

Our application is to assist a player of the game of Bingo. A player has one or more Bingo cards on which they keep track of numbers that have been called out randomly. Each card has 25 different numbers arranged in five rows and five columns. The five columns are labelled B, I, N, G, and O. The range of permissible numbers in a given column are

Column B	1 to 15
Column I	16 to 30
Column N	31 to 45 (and zero)
Column G	46 to 60
Column O	61 to 75

Figure 3-10 shows an example of a Bingo card.

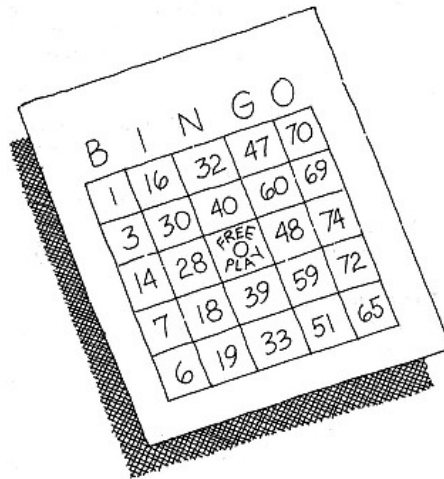


Figure 3-10
A Bingo Card.

When a number is called out, the players look for that number on their cards. If the number is on their card, they cover it with a token. A player wins when he or she has five tokens in one row, in one column, or along a diagonal.

Now that we have discussed our problem at the application level let's look at how we might model this application as a computer program. For now we only are concerned with the design of a data structure (collection class) to represent the Bingo card. We already have described the logical arrangement of data for this class. Here is a specification for a Bingo card that represents our application at the abstract level:

Bingo Card

STRUCTURE: A TABLE OF FIVE ROWS AND FIVE COLUMNS.

OPERATION:

Get Card (File : in)

Function: Gets a Bingo card from a data file.

Input: The file containing the numbers for a Bingo card.

Preconditions: The data file is open. Data in the file is in order by row.

Output: None.

Postconditions: The numbers for the Bingo card are obtained from the file. The Free Play number is covered with a token. All other numbers are cleared of tokens. The Bingo card is valid.

Exceptions: **BAD_NUMBER** Raised if a number in the file is not in the permissible range for the column. The Bingo card is invalid.

Clear Card

Function: Clears all the tokens from a Bingo card and then covers the Free Play number with a token.

Input: None.

Preconditions: The Bingo card is valid (successfully read with Get_Card).

Output: None.

Postconditions: Only the Free Play number is covered with a token.

Cover Number (Column : in; Number : in)

Function: If the given number is on the Bingo card, it is marked.

Input: The Column (B, I, N, G, or O) and the Number.

Preconditions: The Bingo card is valid (successfully read with Get Card).

Output: None.

Postconditions: Column and Number are on the Bingo card and it is marked, or the Bingo card is unchanged.

Exceptions: **BAD_NUMBER** Raised if the Number is not in the permissible range for the Column. The Bingo card is unchanged.

Winner return Boolean

Function: Determines whether the Bingo card is a winner.

Input: None.

Preconditions: The Bingo card is valid (successfully read with `Get_Card`).

Output: True if all the numbers in a row, column or diagonal are marked. Otherwise, returns False.

Postconditions: The Bingo card is unchanged.

We included a comment section on exceptions with two of the Bingo card operations. This additional section serves to separate the postconditions for a normal operation from the postconditions of an abnormal operation. The assertions in the postcondition comments are true if *no* exception is raised by the operation. However, if an exception is raised, it is the assertions in the appropriate exception comment section that are true. Some people prefer to place all assertions in a single postcondition section.

Now that we have seen the application and abstract views of the Bingo card, we are ready to implement it. Our Bingo card specification uses two simple data classes: Bingo numbers and Bingo letters. We could use Ada's predefined classes `Integer` and `Character` for these. However, these predefined classes are very general. You always should use classes that accurately model the application. Specification 3-1 is a package containing classes that accurately model the Bingo numbers and letters.

Specification 3-1— Bingo Definitions

```
package Bingo_Definitions is

-- This package defines two scalar classes used in Bingo games.
--
-- Bingo Numbers and Bingo Letters
--

-- The range of numbers on a Bingo Card
type Bingo_Number is range 0..75;

-- 0 can't be called, it is only for the Free Play square.
subtype Callable_Number is Bingo_Number range 1..75;

-- Associations between Bingo numbers and letters
subtype B_Range is Bingo_Number range 1..15;
subtype I_Range is Bingo_Number range 16..30;
subtype N_Range is Bingo_Number range 31..45;
subtype G_Range is Bingo_Number range 46..60;
subtype O_Range is Bingo_Number range 61..75;
```

```

-- The 5 Bingo letters
type Bingo_Letter is (B, I, N, G, O)

end Bingo_Definitions;

```

We can preinstantiate packages for the input and output of Bingo numbers and letters by compiling the following two library units:

```

with Text_IO;
with Bingo_Definitions;
use Bingo_Definitions;
package Bingo_Number_IO is new Text_IO.Integer_IO (Num => Bingo_Number);

with Text_IO;
with Bingo_Definitions;
use Bingo_Definitions;
package Bingo_Letter_IO is new
    Text_IO Enumeration_IO (Enum => Bingo_Letter);

```

We now look at two different ways to implement the actual Bingo card. Both use Ada's package to encapsulate the implementation of the data structure. Each package specification is a restatement of our abstract view of the data structure. In fact, we frequently use the package specification to express the abstract view. There is really no need for a separate abstract specification like the one given above.

Abstract Data Objects

Abstract Data Object (ADO)

An entity that has a state and a set of operations to examine and change its state. Also called an abstract state machine.

An **abstract data object** (ADO) is used to encapsulate a class when we need only one object from the class. This method is appropriate for our Bingo application if a player is restricted to a single card. With an ADO, the type declarations needed to define the data structure and a variable declaration (to hold one object) are placed *inside the package body*. The package specification contains only the specifications of the operations available to the user and any type declarations required for parameters in these operations. Specification 3-2 contains the declaration for a Bingo card abstract data object.

Specification 3-2— Bingo Card ADO

```

with Text_IO;
with Bingo_Definitions;
use Bingo_Definitions;    -- Make predefined operations on types directly visible

package Bingo_Card_ADO is    -- Abstract Data Object

    BAD_NUMBER : exception;  -- Raised for an invalid Bingo column/number combination

```



```

-----
procedure Get_Card (File : in Text_IO.File_Type);
-- Gets a Bingo card from a data file.

-- Preconditions: The data file is open. Data in the file is in order by row.

-- Postconditions: The numbers for the Bingo card are obtained from the file. The
                  Free Play number is covered with a token. All other numbers
                  are cleared of tokens. The Bingo card is valid.

-- Exception:   BAD_NUMBER   Raised for an invalid Letter/Number.
--                  The Bingo card is invalid.

-----
procedure Clear;
-- Clears the tokens from the Bingo card.

-- Preconditions: The Bingo card is valid (successfully read with Get_Card).

-- Postconditions: Only the Free Play number is covered with a token.

-----
procedure Cover_Number (Column : in Bingo_Letter;
                       Number : in Bingo_Number);
-- If the given number is on the Bingo card, it is marked.

-- Preconditions: The Bingo card is valid (successfully read with Get Card).

-- Postconditions: Column and Number are on the Bingo card and it is marked or
--                  The Bingo card is unchanged.

-- Exceptions:   BAD_NUMBER   Raised for invalid Column/Number.
--                  The Bingo card is unchanged.

-----
function Winner return Boolean;
-- Returns True if all the numbers in a row, column or diagonal of the card are
   marked.

-- Preconditions: The Bingo card is valid (successfully read with Get Card).

-- Postconditions: The Bingo card is unchanged.

end Bingo_Card_ADO;

```

The package body is the actual implementation. In the body we need to decide how to represent the data and how to carry out the operations. A two-dimensional array is one way to represent the data. Each component of the array can contain a number and a Boolean value to indicate whether it is marked. Alternatively, we could use an array of arrays. However, because we must process rows, columns, and diagonals, the two-dimensional array is more appropriate. An array of arrays is appropriate when you need to process only rows. Because we are writing an ADO, the package body must contain a variable to hold one Bingo card. Body 3-1 contains our package body for the Bingo card ADO.

Body 3-1—**Bingo Card ADO Implements Specification 3-2**

```

with Bingo_Number_IO;
package body Bingo_Card_ADO is

    type Component_Type is          -- One component of a Bingo card
        record
            Number   : Bingo_Number;
            Covered  : Boolean := False;
        end record;

    type Row_Range is range 1..5;    -- There are five rows on a Bingo card.

    type Bingo_Card_Array is array (Row_Range, Bingo_Letter) of Component_Type;

    Free_Play_Row : constant Row_Range := 3;
    Free_Play_Col : constant Bingo_Letter := N;

    The_Card : Bingo_Card_Array; -- The Bingo card object (used globally)

    -----
    procedure Clear is

        -- This procedure removes all the markers from the Bingo Card except on the free
        -- play.

    begin
        for Row in Row_Range loop
            for Column in Bingo_Letter loop
                The_Card (Row, Column).Covered := False;
            end loop;
        end loop;
        The_Card (Free_Play_Row, Free_Play_Col) := (Number => 0, Covered => True);
    end Clear;

    -----
    function Valid_Number (Column : in Bingo_Letter;
                           Number : in Bingo_Number) return Boolean is

        -- This function returns True if the Bingo number given -- is valid for the given Bingo column.

    Result : Boolean;

    begin
        case Column is
            when B => Result := Number in B_Range;
            when I => Result := Number in I_Range;
            when N => Result := (Number in N_Range) or (Number = 0);
        end case;
    end Valid_Number;
end Bingo_Card_ADO;

```



```
-- Determines whether or not all the numbers in Row are covered

Column : Bingo_Letter; -- The column in the row being checked

begin -- Check_One_Row
  Column := Bingo_Letter'First;
  Column_Loop: -- Each iteration, check one number in the row.
  loop
    -- Exit when we find an uncovered number or we reach end of the row.
    exit Column_Loop when not The_Card(Row, Column).Covered or
      Column = Bingo_Letter'Last;
    Column := Bingo_Letter'Succ (Column);
  end loop Column_Loop;
  -- If we didn't exit the loop because of an uncovered number,
  -- all the numbers in the row are covered.
  All_Covered := The_Card(Row, Column).Covered;
end Check_One_Row;
```

```
-----
function Row_Winner return Boolean is
-- Returns True if all the numbers in a row are marked
```

```
Row : Row_Range; -- The row being checked
Found_Winner : Boolean; -- Result of a row check

begin -- Row Winner
  Row := 1;
  Row_Loop: -- Each iteration, check one row.
  loop
    Check_One_Row (Row => Row,
      All_Covered => Found_Winner);
    -- Exit when we found a winner or run out of rows
    exit Row_Loop when Found_Winner or Row = Row_Range'Last;
    Row := Row + 1;
  end loop Row_Loop;
  return Found_Winner;
end Row_Winner;
```

```
-----
procedure Check_One_Column (Column : in Bingo_Letter;
  All_Covered : out Boolean) is
-- Determines whether or not all the numbers in Column are covered
```

```
Row : Row_Range; -- The row in the column being checked

begin -- Check_One_Column
  Row := 1;
  Row_Loop: -- Each iteration, check one number in the Column.
  loop
    -- Exit when we find an uncovered number or reach end of column.
    exit Row_Loop when not The_Card(Row, Column).Covered or
      Row = Row_Range'Last;
    Row := Row + 1;
  end loop Row_Loop;
```

```

-- If we didn't exit the loop because of an uncovered number,
-- all the numbers in the column are covered.
All_Covered := The_Card(Row, Column).Covered;
end Check_One_Column;

```

```

-----
function Column_Winner return Boolean is
-- Returns True if all the numbers in a column are marked

```

```

    Column      : Bingo_Letter; -- The column being checked
    Found_Winner : Boolean;      -- Result of a column check

```

```

begin
    Column := Bingo_Letter'First;
    Column_Loop: -- Each iteration, check one column.
    loop
        Check_One_Column (Column => Column,
                          All_Covered => Found_Winner);
        -- Exit when we found a winner or run out of columns
    exit Column_Loop when Found_Winner or
                          Column = Bingo_Letter'Last;
        Column := Bingo_Letter'Succ (Column);
    end loop Column_Loop;
    return Found_Winner;
end Column_Winner;

```

```

-----
function Diagonal_Winner return Boolean is
-- Returns True if all the numbers in a diagonal are marked

```

```

    Column      : Bingo_Letter; -- The column being checked
    Row         : Row_Range;    -- The row being checked
    Found_Loser : Boolean;       -- Result of a number check

```

```

begin
    -- First diagonal
    Row := 1;
    Column := Bingo_Letter'First;
    loop
        Found_Loser := not The_Card(Row, Column).Covered;
        -- Exit if we find an uncovered number (loser) or reach the end of the
        -- diagonal.
    exit when Found_Loser or Row = Row_Range'Last;
        Row := Row + 1;
        Column := Bingo_Letter'Succ (Column);
    end loop;

    if Found_Loser then -- Check the other diagonal
        Row := 1;
        Column := Bingo_Letter'Last;
        loop
            Found_Loser := not The_Card(Row, Column).Covered;
            -- Exit if we find an uncovered number (loser) or reach the
            -- end of the diagonal.

```

```

        exit when Found_Loser    or Row = Row_Range'Last;
        Row := Row + 1;
        Column := Bingo_Letter'Pred (Column);
    end loop;
end if;

    return not Found_Loser;
end Diagonal_Winner;

```

```

-----
function Winner return Boolean is
begin
    return Row_Winner    or else
           Column_Winner or else
           Diagonal_Winner;
end Winner;

```

```

end Bingo_Card_ADO;

```

The variable `The_Card` is our model of an actual Bingo card. Note that it is declared and used globally by the subprograms in the package body. You may have learned (correctly) that the use of global variables should be avoided to prevent side effects. The viability of an abstract data object, however, depends upon all the operations accessing the global object variable. The Bingo card is intentionally left out of the parameter lists of the operations to prevent the programmer who is using the package from accessing the details of the card. In this case, the benefits of encapsulation outweigh any disadvantages of using a global variable. The global variable's scope is restricted to a single package body; it could be called a *local global variable*. Figure 3-11 illustrates how an ADO encapsulates this variable so that clients using the Bingo card ADO can only access `The_Card` indirectly through the operations supplied in the package specification.

Some of the algorithms to implement the operations specified in our abstract view of the Bingo card are quite lengthy. Because we have encapsulated these algorithms, we can use the Bingo card object without understanding these algorithms in the same way we use an array without understanding its implementation.

Finally, note that this package body contains procedures that are not declared in the package specification as operations for the Bingo card. These procedures are the implementation of modules from a top-down design of each of the Bingo card's operations. Although we could have written the Bingo card's operations as single subprograms, the additional levels of abstraction provided by the subprograms simplify our logic. Because these procedures are completely inside the Bingo card package body (they are not declared in the package specification), they are not visible to the package clients.

Abstract Data Types

The major advantage of using an abstract data object (ADO) to implement an object class is that the object is fully encapsulated. The user of the package is unable to see the details of the object. The object may only be modified through the operations declared in the package specification. The major disadvantage of using an ADO to implement an object class is that it models a *single object*. It cannot be used for an arbitrary number of objects. For example, the ADO we developed in the last section cannot be used for a player who

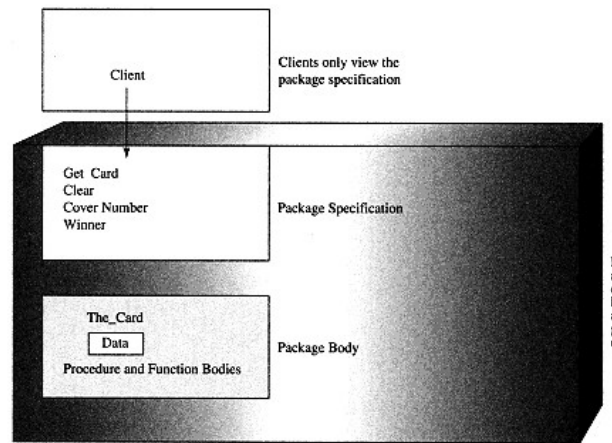


Figure 3-11
The Bingo Card ADO

plays several different Bingo cards in one game. We could duplicate our object package for each object. If we played three cards, we could have three identical packages called `Bingo_Card_1`, `Bingo_Card_2`, and `Bingo_Card_3`. This requires considerable work on our part; imagine if a problem required 1,000 Bingo cards.

Abstract Data Type (ADT)

A class of data objects with a defined set of properties and a set of operations that process the data objects while maintaining the properties.

The **abstract data type** (ADT) allows us to create as many objects of a class as we need. We do not have to duplicate any code to accomplish this. Specification 3-3 shows how we can implement our Bingo card class as an ADT.

Specification 3-3— Bingo Card ADT

```

with Text_IO;
with Bingo_Definitions;
use Bingo_Definitions;
package Bingo_Card_ADT is      -- Abstract Data Type

    type Card_Type is private; -- The Bingo Card Class

    BAD_NUMBER : exception; -- Raised for an invalid Bingo column/number combination

    -----
    procedure Get_Card (File : in Text_IO.File_Type;
                       Card : out Card_Type);
    -- Gets a Bingo card from a data file.
  
```

```

-- Preconditions:  The data file is open. Data in the file is in order by row.

-- Postconditions: The numbers for the Bingo card are obtained from the file. The
--                 Free Play number is covered with a token. All other numbers
--                 are cleared of tokens.

-- Exception:     BAD_NUMBER    Raised for an invalid Letter/Number.

-----
procedure Clear(Card : in out Card_Type);
-- Clears the tokens from the Bingo card.

-- Preconditions:  The Bingo card is valid (successfully read with Get_Card).

-- Postconditions: Only the Free Play number is covered with a token.

-----
procedure Cover_Number (Card   : in out Card_Type;
                       Column : in   Bingo_Letter;
                       Number  : in   Bingo_Number);
-- If the given number is on the Bingo card, it is marked.

-- Preconditions:  The Bingo card is valid (successfully read with Get Card).

-- Postconditions: If the given Column and Number is on the Bingo card, it is
--                 marked. Otherwise the Bingo card is unchanged.

-- Exceptions:     BAD_NUMBER    Raised for invalid Column/Number.

-----
function Winner (Card : in Card_Type) return Boolean;
-- Returns True if all the numbers in a row, column, or diagonal
-- of the card are marked.

-- Preconditions:  The Bingo card is valid (successfully read with Get Card).

-- Postconditions: The Bingo card is unchanged.

private

type Component_Type is          -- One component of a Bingo card
  record
    Number : Bingo_Number;
    Covered : Boolean := False;
  end record;

type Row_Range is range 1..5;   -- There are five rows on a Bingo card.

type Card_Type is array (Row_Range, Bingo_Letter) of Component_Type;

end Bingo_Card_ADT;

```


Private Type

A type declared in the visible part of a package declaration whose details (record fields, array index, and component types, and so on) are declared in the private part of the package declaration; the type may be used outside of the package, but its internal components may not.

There are three differences between this package specification and the one used for the ADO. First, a new type, `Card_Type`, is declared. This type *is* the Bingo card class and is available to the user of this package. Users can declare as many Bingo cards as they need, even an array of Bingo cards. Second, every operation now contains a parameter for a Bingo card. To see if one of our cards is a winner, we need to pass that particular card to the function `Winner`. Third, we have used a **private type**. The package specification is divided into two portions. Everything above the reserved word `private` is accessible to a user of this package. Everything below `private` is inaccessible. This is similar to the library example discussed earlier in this chapter.

`Card_Type` is declared as `private`. This means that a programmer using this package can declare objects (variables and constants) of this class. However, the operations allowed with the objects are restricted to the operations given in the package, equality testing (`=` and `/=`), and assignment (`:=`). The Ada compiler encapsulates the class by enforcing these restrictions.

We also could have declared `Card_Type` as `limited private`. Such a declaration limits the operations on the type even further. Only the operations given in the package may be used with limited private objects. Limited private objects cannot even be assigned with the `:=` operator. In later chapters we look at uses for this highly restrictive declaration.

The actual declaration for a `private` type is given in the private section of the package specification. We can *see* that the Bingo card class is implemented as a two-dimensional array—private does not mean invisible. However, the declarations in the private part of this package specification are unavailable outside of package `Bingo_Card`. This means, for example, that a user is unable to use the subscripts of the array or the `Covered` field of the record to check the status of a Bingo card. To access the card, users *must* use the operations defined in the package.

The private section is much like the closed stacks of our library example. Patrons of the library may be able to see the stacks through the glass window behind which a librarian sits. However, they can only access the books through the operations defined by the library staff.

There are only three minor changes we need to make to the package body we wrote for the ADO package implementation. First we must delete all the declarations that are now in the private part of the ADT package specification. Second, we must delete the declaration of the global variable `The_Card`. And third, we must replace all references to the global variable with parameter references. Body 3-2 incorporates these three changes.

Body 3-2—**Bingo Card ADT Implements Specification 3-3**

```
with Bingo_Number_IO;
package body Bingo_Card_ADT is

    Free_Play_Row : constant Row_Range := 3;
    Free_Play_Col : constant Bingo_Letter := N;
```

```

-----
procedure Clear (Card : in out Card_Type) is

-- This procedure removes all the markers from the Bingo Card.

begin
  for Row in Row_Range loop
    for Column in Bingo_Letter loop
      Card (Row, Column).Covered := False;
    end loop;
  end loop;
  Card (Free_Play_Row, Free_Play_Col) := (Number => 0, Covered => True);
end Clear;

-----

function Valid_Number (Column : in Bingo_Letter;
                      Number : in Bingo_Number) return Boolean is

-- This function returns True if the Bingo number given
-- is valid for the given Bingo column.

  Result : Boolean;

begin
  case Column is
    when B => Result := Number in B_Range;
    when I => Result := Number in I_Range;
    when N => Result := (Number in N_Range) or (Number = 0);
    when G => Result := Number in G_Range;
    when O => Result := Number in O_Range;
  end case;
  return Result;
end Valid_Number;

-----

procedure Get_Card (File : in Text_IO.File_Type;
                  Card : out Card_Type) is

  Number : Bingo_Number; -- A Bingo number

begin
  for Row in Row_Range loop
    for Column in Bingo_Letter loop
      Bingo_Number_IO.Get (File => File, Item => Number);
      if Valid_Number (Column, Number) then
        Card (Row, Column) := (Number => Number, Covered => False);
      else
        raise BAD_NUMBER;
      end if;
    end loop;
  end loop;
  Card (Free_Play_Row, Free_Play_Col).Covered := True;
end Get_Card;

-----

```

```

procedure Cover_Number (Card   : in out Card_Type;
                        Column : in   Bingo_Letter;
                        Number  : in   Bingo_Number) is

    Row : Row_Range; -- Loop control variable

begin
    if not Valid_Number (Column, Number) then
        raise BAD_NUMBER;
    end if;

    -- Search the Column for the given Number.
    Row := 1;
    Search_Loop: -- Each iteration, one number in Column is checked.
    loop
        -- Exit when we find the number or reach the end of the column.
        exit Search_Loop when Number = Card (Row, Column).Number or
            Row = Row_Range'Last;

        Row := Row + 1;
    end loop Search_Loop;
    -- If we found the number, cover it.
    if Number = Card (Row, Column).Number then
        Card (Row, Column).Covered := True;
    end if;
end Cover_Number;

-----
procedure Check_One_Row (Card       : in Card_Type;
                        Row         : in Row_Range;
                        All_Covered : out Boolean) is
    -- Determines whether or not all the numbers in Row are covered

    Column : Bingo_Letter; -- The column in the row being checked

begin -- Check_One_Row
    Column := Bingo_Letter'First;
    Column_Loop: -- Each iteration, check one number in the row.
    loop
        -- Exit when we find an uncovered number or we reach end of the row.
        exit Column_Loop when not Card(Row, Column).Covered or
            Column = Bingo_Letter'Last;

        Column := Bingo_Letter'Succ (Column);
    end loop Column_Loop;
    -- If we didn't exit the loop because of an uncovered number,
    -- all the numbers in the row are covered.
    All_Covered := Card(Row, Column).Covered;
end Check_One_Row;

-----
function Row_Winner (Card : in Card_Type) return Boolean is
    -- Returns True if all the numbers in a row are marked.

    Row           : Row_Range; -- The row being checked
    Found_Winner  : Boolean;    -- Result of a row check

```

```

begin -- Row Winner
  Row := 1;
  Row_Loop: -- Each iteration, check one row.
  loop
    Check_One_Row (Card      => Card,
                  Row        => Row,
                  All_Covered => Found_Winner);
    -- Exit when we found a winner or run out of rows.
    exit Row_Loop when Found_Winner or Row = Row_Range'Last;
    Row := Row + 1;
  end loop Row_Loop;
  return Found_Winner;
end Row_Winner;

-----

procedure Check_One_Column (Card      : in Card_Type;
                           Column     : in Bingo_Letter;
                           All_Covered : out Boolean) is
-- Determines whether or not all the numbers in Column are covered

  Row : Row_Range; -- The row in the column being checked

begin -- Check_One_Column
  Row := 1;
  Row_Loop: -- Each iteration, check one number in the Column.
  loop
    -- Exit when we find an uncovered number or reach end of column.
    exit Row_Loop when not Card(Row, Column).Covered or
                      Row = Row_Range'Last;
    Row := Row + 1;
  end loop Row_Loop;
  -- If we didn't exit the loop because of an uncovered number,
  -- all the numbers in the column are covered.
  All_Covered := Card(Row, Column).Covered;
end Check_One_Column;

-----

function Column_Winner (Card : in Card_Type) return Boolean is
-- Returns True if all the numbers in a column are marked.

  Column      : Bingo_Letter; -- The column being checked
  Found_Winner : Boolean;      -- Result of a column check

begin -- Column_Winner
  Column := Bingo_Letter'First;
  Column_Loop: -- Each iteration, check one column.
  loop
    Check_One_Column (Card      => Card,
                    Column     => Column,
                    All_Covered => Found_Winner);
    -- Exit when we found a winner or run out of columns.
    exit Column_Loop when Found_Winner or
                      Column = Bingo_Letter'Last;
    Column := Bingo_Letter'Succ (Column);
  end loop Column_Loop;

```

```

    return Found_Winner;
end Column_Winner;

-----
function Diagonal_Winner (Card : in Card_Type) return Boolean is
-- Returns True if all the numbers in a diagonal are marked

    Column      : Bingo_Letter; -- The column being checked
    Row         : Row_Range;    -- The row being checked
    Found_Loser : Boolean;       -- Result of a number check

begin
    -- First diagonal
    Row := 1;
    Column := Bingo_Letter'First;
    loop
        Found_Loser := not Card(Row, Column).Covered;
        -- Exit if we find an uncovered number (loser) or reach the
        -- end of the diagonal.
        exit when Found_Loser or Row = Row_Range'Last;
        Row := Row + 1;
        Column := Bingo_Letter'Succ (Column);
    end loop;

    if Found_Loser then -- Check the other diagonal.
        Row := 1;
        Column := Bingo_Letter'Last;
        loop
            Found_Loser := not Card(Row, Column).Covered;
            -- Exit if we find an uncovered number (loser) or reach the
            -- end of the diagonal.
            exit when Found_Loser or Row = Row_Range'Last;
            Row := Row + 1;
            Column := Bingo_Letter'Pred (Column);
        end loop;
    end if;
    return not Found_Loser;
end Diagonal_Winner;

-----
function Winner (Card : in Card_Type) return Boolean is
begin
    return Row_Winner (Card)      or else
           Column_Winner (Card)   or else
           Diagonal_Winner (Card);
end Winner;

end Bingo_Card_ADT;

```

Note that `Card` is a parameter supplied by the client. Clients can declare as many variables of `Card_Type` as they need. Figure 3-12 shows a client with two Bingo card variables. Although the data is stored within the client's variables, its private type ensures that this data is only manipulated by the operations given in the package specification.

This situation is analogous to Ada's predefined types. You can declare as many integer variables as you like, but the operations that you can perform on them are restricted to those integer operations defined in package `Standard` (Appendix G).

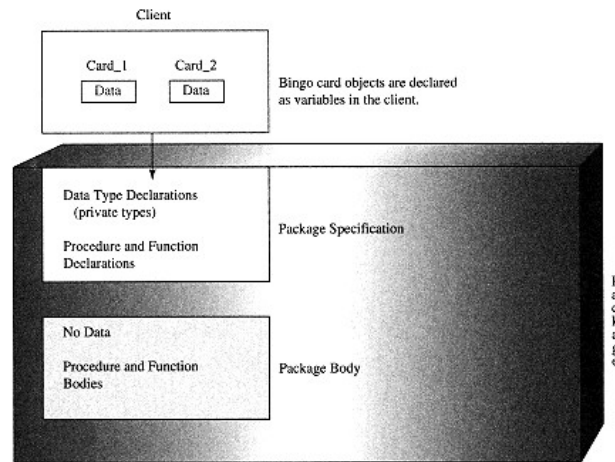


Figure 3-12
The Bingo Card ADT with Two Bingo Card Objects

Summary

We have discussed how data can be viewed from multiple perspectives, and we have seen how Ada encapsulates the implementation of its predefined types and allows us to encapsulate our own classes.

As we create data structures (collection classes), using built-in data types such as arrays and records to implement them, we see that there are actually many levels of data abstraction. The abstract view of an array might be seen as the implementation level of the programmer-defined data structure `List`, which uses an array to hold its elements. At the abstract level, we do not access the elements of `List` through their array indexes but through a set of accessing operations defined especially for objects of `List` type. Moving up a level, we might see the abstract view of `List` as the implementation level of another programmer-defined data type, `Product Inventory`.

What do we gain by separating the views of the data? First, we reduce complexity at the higher levels of the design, making the program easier to understand. Second, we make the program more easily modifiable: The implementation can be completely changed without affecting the program that uses the data structure. We use this advantage in this text, developing various implementations of the same object class in different chapters. Third, we develop software that is *reusable*: The structure and its accessing operations can be used by other programs, for completely different applications, as long as the correct interfaces are maintained. You saw in the first two chapters of this book that the design, implementation, and verification of high quality computer software is a very laborious process. Being able to reuse pieces that are already designed, coded, and tested cuts down on the amount of work we have to do.

Application or user view	Logical or abstract view	Implementation view
Product Inventory	List	Array
List	Array	Row major access function
Array	Row major access function	32-Bit words on IBM Power PC

Perspectives on Data

In the chapters that follow we extend these ideas to build other collection classes that Ada does not provide: sets, strings, stacks, queues, lists, trees, and graphs. We consider these classes from the abstract view: What is our logical picture of the data, and what operations can we use to create, assign to, and manipulate elements in the data structure? We express our abstract view as either a package specification for an abstract data object (ADO) or an abstract data type (ADT).

Next, we take the application view of the data, using the data structures we have developed in some real programs.

Finally, we change hats and turn to the implementation view of the class. We consider the Ada type declarations that represent the data structure, as well as the design of procedures and functions that implement the specifications of the abstract view. Data structures can be implemented in more than one way, so we often look at alternative representations and methods for comparing them.

Case Study:

Bingo Games—How Long Should They Take?

How many numbers are called in a game of Bingo? That depends on the order in which the numbers are called and how many cards are played. Those versed in probability theory can calculate the expected number of *calls* for a given number of Bingo cards. Computer simulation is another way to learn more about this problem. We can write a program to stimulate thousands of Bingo games and determine the average number of *calls*. Here is a specification for such a program:

Bingo Simulation

Function: To learn about the number of number calls made in a typical Bingo game.

Input: The number of games to simulate (entered from console).

The number of cards to use in each game (entered from console).

The name of the file containing the Bingo card numbers (entered from console).

The numbers on the cards (stored in a data file).

Output: **The average number of calls in the simulated games.**

The maximum number of calls in the simulated games.

The minimum number of calls in the simulated games.

First, let's analyze what objects are in this problem. The input and output values are numbers and a file name. These can be modeled by Ada's built in types `Natural` and `String`. We already have discussed the Bingo card object which is obviously useful for this program. We still need a way of calling the Bingo numbers. This process usually is done with a basket or bin from which the numbers are drawn at random. There are two operations for this object: *Load* all of the Bingo numbers and *Draw* a number. Because we only need one Bingo basket object, we can use an abstract data object (ADO). Its specification is shown in Specification 3-4.

Specification 3-4— Bingo Basket ADO

```
with Bingo_Definitions;
package Bingo_Basket is

-- This package implements a single Bingo Basket,
-- an object from which Bingo Numbers are drawn.

-- Structure: An unordered collection of Bingo numbers.

EMPTY : exception; -- Raised on attempt to draw a number from an empty basket.
-----
procedure Load;

-- Places all of the numbers into the basket.

-- Preconditions: None.

-- Postconditions: All 75 Bingo numbers are in the basket.

-----
procedure Draw (Letter : out Bingo_Definitions.Bingo_Letter;
               Number : out Bingo_Definitions.Callable_Number);

-- Randomly selects a number from the basket.

-- Preconditions: None.

-- Postconditions: Number contains a Bingo number not previously drawn.
--                 Letter contains the Bingo Letter corresponding to Number.
--                 Number is removed from the basket.

-- Exceptions:     EMPTY is raised if the basket is empty.
--                 The basket remains empty.

end Bingo_Basket;
```


We need more than one Bingo card for our simulation so we use our ADT version of the card. Because we require a group of these cards, we need to consider the design of another collection class for a group of Bingo cards. The only operation required for this group of cards is access to individual cards. The predefined array class fulfills this requirement, so we can implement our group of Bingo cards with an array.

Now that we specified the structure and operations of the objects for our program, we need to determine how the objects interact. A real Bingo game is coordinated by a number caller, who draws numbers from the basket, announces the numbers, and waits for someone to say "BINGO." We can specify another object for the number caller. Such an object often is called a *director object* as it directs the actions of the other objects in our program. The only operation of a director object is to carry out the function of the program; in our case, to simulate a number of Bingo games. We typically implement the director object's single operation as the main program.* As with all object *operations*, we design the main program using functional decomposition. Having already defined the Bingo card and Bingo basket objects makes this design easier to construct. Here is our algorithm:

Bingo Simulation Program

Level 0

```
Get number of games
Get number of cards
Play all the games
Report the results
```

Play All Games (Statistics : out)

Level 1

```
Get all the cards
for Game in 1..Number of Games loop
    Play one game
    Add Draws to Total
    Check for extremes (minimum and maximum)
end loop
Calculate average
```

Get All Cards (Cards : out)

Level 2

```
Open input file
for Index in 1..Number of Cards
    Get Bingo Card(Index) from the file
end loop
Close input file
```

*More complex problems may require several director objects. Multiple director objects are best implemented as Ada tasks, a topic beyond the scope of this text.

Play One Game (Cards: in out; Draws : out)

```

Clear all markers from Cards
Bingo_Basket.Load
Draws := 0
loop
    Draws := Draws + 1
    Draw a Number from the Bingo Basket
    Check Cards for a winner
    exit when there is a winning card
end loop

```

Check Cards for a Winner (Cards : in out; Winner out);*Level 3*

```

Index := 1
loop
    Cover_Number for Card(Index)
    exit when found a winner or processed last card
    Index := Index + 1;
end loop

```

The resulting program is shown as Program 3-1.

**Program 3-1—
Bingo Simulation**

```

with Bingo_Definitions;
use   Bingo_Definitions;
with Bingo_Card_ADT;
with Bingo_Basket;
with Integer_IO;
with Float_IO;
with Text_IO;

procedure Bingo_Simulation is

-- This program is used to learn about the number of draws made in a typical Bingo
-- game.

-- Types for Statistics
type Stat_Rec is
    record
        Average : Float;    -- Average number of calls
        Minimum : Natural;  -- Minimum number of calls
        Maximum : Natural;  -- Maximum number of calls
    end record;

-- Type for a collection of Bingo Cards
type Bingo_Card_Array is array (Positive range <>) of Bingo_Card_ADT.Card_Type;

subtype File_String is String (1..20);

-- Variables for main program
Num_Games   : Positive;    -- The number of games to simulate

```

```

Num_Cards   : Positive;      -- The number of cards in each game
Bingo_Stats : Stat_Rec;      -- The resulting statistics
File_Name   : File_String;   -- Name of file containing Bingo card data
Last        : Natural;       -- Number of characters in File_Name

```

```

-----
procedure Check_Cards (Letter : in      Bingo_Letter;
                      Number : in      Bingo_Number;
                      Cards  : in out Bingo_Card_Array;
                      Winner : out Boolean) is
-- Covers the number drawn and checks cards for a winner
  Card_Index   : Positive;    -- Loop control variable
  Found_Winner : Boolean;      -- Local copy of out parameter Winner

```

```

begin -- Check_Cards
  Card_Index := Cards'First;
  Card_Loop:
  loop -- Each iteration one card is checked
    Bingo_Card_ADT.Cover_Number (Card => Cards(Card_Index),
                                Column => Letter,
                                Number => Number);
    Found_Winner := Bingo_Card_ADT.Winner (Cards(Card_Index));
    exit Card_Loop when Found_Winner or Card_Index = Cards'Last;
    Card_Index := Card_Index + 1;
  end loop Card_Loop;
  Winner := Found_Winner; -- Copy results to out parameter
end Check_Cards;

```

```

-----
procedure Play_One_Game (Cards : in out Bingo_Card_Array;
                       Draws : out Positive) is
-- This procedure plays one game of Bingo using the cards passed in.
  Num_Draws   : Natural := 0; -- Local copy of out parameter Draws
  Letter      : Bingo_Letter; -- The letter and
  Number      : Bingo_Number; -- number drawn
  Found_Winner : Boolean;      -- True when a winning card is found

```

```

begin -- Play_One_Game
  Bingo_Basket.Load; -- Put all of the numbers into the basket
  for Index in Cards'Range loop -- Clear markers from all cards
    Bingo_Card_ADT.Clear (Card => Cards(Index));
  end loop;
  -- Play the game
  Draw_Loop:
  loop -- Each iteration, one draw is processed
    Num_Draws := Num_Draws + 1;
    Bingo_Basket.Draw (Letter => Letter, Number => Number);
    -- Cover the number on all the cards and check for a winner
    Check_Cards (Letter => Letter, Number => Number,
                 Cards => Cards, Winner => Found_Winner);
    exit Draw_Loop when Found_Winner;
  end loop Draw_Loop;
  Draws := Num_Draws; -- Copy result to out parameter
end Play_One_Game;

```

```

procedure Get_Cards (Cards      : out Bingo_Card_Array;
                    File_Name : in String) is
-- This procedure reads all of the Bingo cards from the data file.

-- Preconditions:  The data file exists
--                Bingo card data in the file is valid

    Card_Data : Text_IO.File_Type; -- File containing Bingo card data

begin -- Get_Cards
    Text_IO.Open (File => Card_Data,
                  Name => File_Name,
                  Mode => Text_IO.In_File);
    for Index in Cards'Range loop
        Bingo_Card_ADT.Get_Card (File => Card_Data,
                                Card => Cards(Index));
    end loop;
    Text_IO.Close (Card_Data);
end Get_Cards;

-----
procedure Play_All_Games (Num_Games : in Positive;
                         Num_Cards : in Positive;
                         File_Name : in String;
                         Stats      : out Stat_Rec) is
-- This procedure simulates Bingo games and keeps track of statistics.
-- Constrained array type just large enough for the required number of cards.
    subtype Card_Array is Bingo_Card_Array (1..Num_Cards);

    Cards      : Card_Array; -- All the Bingo cards
    Draws      : Natural;    -- Draws for one Bingo game
    Total_Draws : Natural := 0; -- Total number of draws for all games
    Min_So_Far  : Natural := Natural'Last; -- For determining
    Max_So_Far  : Natural := Natural'First; -- draw extremes

begin -- Play_All_Games
    -- Get the Bingo cards
    Get_Cards (Cards => Cards, File_Name => File_Name);

    -- Play the games
    for Game_Count in 1.. Num_Games loop
        Play_One_Game (Cards => Cards, Draws => Draws);
        Total_Draws := Total_Draws + Draws; -- Update number of draws
        -- Check for new extreme values
        if Draws > Max_So_Far then
            Max_So_Far := Draws;
        end if;
        if Draws < Min_So_Far then
            Min_So_Far := Draws;
        end if;
    end loop;
end Play_All_Games;

```

```

-- Fill in Stats
Stats.Average := Float(Total_Draws) / Float(Num_Games);
Stats.Minimum := Min_So_Far;
Stats.Maximum := Max_So_Far;
end Play_All_Games;

-----
begin -- Bingo Simulation

-- Get the simulation parameters
Text_IO.Put ("Enter number of Bingo games to simulate: ");
Integer_IO.Get (Num_Games);
Text_IO.Put ("Enter number of Bingo cards to use: ");
Integer_IO.Get (Num_Cards);
Text_IO.Skip_Line;
Text_IO.Put ("Enter name of file containing the Bingo Card data: ");
Text_IO.Get_Line (Item => File_Name, Last => Last);

-- Carry out the simulation
Play_All_Games (Num_Games => Num_Games,
                Num_Cards => Num_Cards,
                File_Name => File_Name(1..Last),
                Stats      => Bingo_Stats);

-- Report the results
Text_IO.New_Line (2);
Text_IO.Put ("Average number of draws was ");
Float_IO.Put (Item => Bingo_Stats.Average, Fore => 3, Aft => 2, Exp => 0);
Text_IO.New_Line;
Text_IO.Put ("Minimum number of draws was ");
Integer_IO.Put (Item => Bingo_Stats.Minimum, Width => 3);
Text_IO.New_Line;
Text_IO.Put ("Maximum number of draws was ");
Integer_IO.Put (Item => Bingo_Stats.Maximum, Width => 3);
Text_IO.New_Line;
end Bingo_Simulation;

```

We implemented the body of `Bingo_Card` earlier in this chapter. Only the body of `Bingo_Basket` remains to be implemented. To do that, you need to understand sets, the topic of Chapter 4. We continue this case study there.

Exercises

1. Name three different perspectives from which we can view an object class. Using the logical data structure "a list of student academic records," give examples of what each perspective might tell us about the data.
2. Consider the class `Grocery Store`.
 - (a) At the application level, describe `Grocery Store`.
 - (b) At the abstract level, what grocery store operations might be defined for the customer?
 - (c) Specify (at the abstract level) the operation `Check Out`.

(d) Write an algorithm (at the implementation level) for the operation Check Out.

(e) Explain how parts (c) and (d) represent information hiding.

3. What composite types are predefined in the Ada language?

4. Describe the accessing function of an Ada one-dimensional array at the abstract level.

5. What operations does Ada define for the array data type?

6. You are given the following declarations:

```
type B_Array is array (0..4) of Boolean;
type Grade_Type is (A, B, C, D, F);
type N_Array is array (Grade_Type) of Natural;
X : B_Array := (0..2 => True, 3..4 => False);
Y : B_Array := (2 | 3 => True, others => False);
G : N_Array := (24, 35, 41, 33, 18);
H : N_Array := (24, 35, 40, 32, 19);
```

What is the value of each of the following expressions? (See Appendix I for explanations of any attributes you are not familiar with.)

(a) X'First	(k) G'First
(b) X'Last	(l) G'Last
(c) X'Length	(m) G'Length
(d) X'Range	(n) G'Range
(e) X(2) and Y(2)	(o) G < H
(f) X and Y	(p) G(A..B) < H(D..F)
(g) X(0..2) & Y(2..3)	(q) G(A..B) & H(C..F)
(h) X(1) = Y(1)	(r) G(F..F) & G(D..D) & G(C..C) & G(B..B) & G(A..A)
(i) X = Y	(s) G(A..B) & H(C..F) < H(A..B) & G(C..F)
(j) X < Y (Hint: True is greater than False)	

7. (a) Declare a one-dimensional array, Name, that contains 20 characters indexed by Positive values.

(b) If each character takes one "cell" in memory and the base address of Name is 1000, what is the address of the cell referenced in this assignment statement?

```
Name (10) := 'A';
```

Use the following declarations for exercises 8 and 9:

```
type Month_Type is (January, February, March, April,
                   May, June, July, August, September,
                   October, November, December);
subtype Year_Range is Integer range 1980..1999;
```

```
type Weather_Rec is
  record
    Avg_Hi_Temp : Integer;
    Avg_Lo_Temp : Integer;
    Actual_Rain : Float;
    Record_Rain : Float;
  end record;
```

Assume that an integer takes one cell in memory, a float number takes two cells, and that the fields are in contiguous memory locations with no gaps.

8. (a) Declare a one-dimensional array type, `Weather_List_Type`, of `Weather_Rec` components, indexed by `Month_Type`. Declare a variable, `Yearly_Weather`, of `Weather_List_Type`.

(b) Assign the value 1.05 to the actual rainfall field of the July record in `Yearly_Weather`.

(c) If the base address of `Yearly_Weather` is 200, what is the address of the field that you assigned in (b) above?

9. (a) Declare a two-dimensional array, `Decade_Weather`, of `Weather_Rec` components, indexed by `Month_Type` in the first dimension and by `Year_Range` in the second dimension.

(b) Draw a picture of `Decade_Weather`.

(c) Assign the value 26 to the `Avg_Lo_Temp` field of the March, 1989, record.

10. (a) Define a three-dimensional array at the logical level.

(b) Suggest some applications for three-dimensional arrays.

Use the following declarations for exercises 11–13:

```
subtype Name_Type is String (1..10);
subtype GPA_Type is Float range 0.0..4.0;
type Student_Rec is
  record
    First_Name   : Name_Type;
    Last_Name    : Name_Type;
    ID           : Positive;
    GPA          : GPA_Type;
    Current_Hours : Natural;
    Total_Hours  : Natural;
  end record;
```

Assume that an integer or character takes one cell in memory, a float number takes two cells, and that the fields are in contiguous memory locations with no gaps.

11. Construct a field-length-offset table (as in Figure 3-5) for `Student_Rec`.

12. If the base address of `Student` (a variable of `Student_Rec`) is 100, what address does the compiler generate as the target of the following assignment statement?

```
Student.GPA := 3.87;
```

13. How much space does the compiler reserve for `Student_List`, declared below?

```
type List_Type is ARRAY (1..100) of Student_Rec;

Student_List : List_Type;
```

14. Indicate which predefined Ada types would most appropriately model each of the following (more than one may be appropriate for each):

- (a) a chessboard
- (b) information about a single product in an inventory-control program
- (c) a list of famous quotations
- (d) the casualty figures (number of deaths per year) for highway accidents in Texas from 1954 to 1974
- (e) the casualty figures for highway accidents in each of the states from 1954 to 1974

- (f) the casualty figures for highway accidents in each of the states from 1954 to 1975, subdivided by month
 - (g) an electronic address book (name, address, and phone information for all your friends)
 - (h) a collection of hourly temperatures for a 24-hour period
15. What Ada construct is used to encapsulate data structures (collection classes)?
16. Explain the difference between an abstract data object (ADO) and an abstract data type (ADT).
17. (a) How is the programmer using an ADO package prevented from accessing the details of the object?
- (b) How do we prevent the programmer using an ADT package from accessing the details of the type defining the class?
 - (c) What is the advantage of an abstract data type over an abstract data object?
 - (d) What is the advantage of an abstract data object over an abstract data type?
18. *True or False?*
- (a) The details of a private type can be *seen* by a programmer using an ADT package.
 - (b) The details of a private type may be *accessed* by a programmer using an ADT package.
19. Explain the difference between a private type and a limited private type.
20. Use the Bingo card ADT, Specification 3-3 of the chapter, to perform the following:
- (a) Declare two Bingo card objects (variables) called `My_Card` and `Your_Card`.
 - (b) Declare an unconstrained array (type) of Bingo cards.
 - (c) Write a specification for a function, `Num_Winners`, that is given an unconstrained array of Bingo cards and returns how many of them are winners.
 - (d) Implement the function `Num_Winners`.
21. Use the following declarations for "square" matrices in your answers for the following problems.
- ```

type Matrix is array (Positive range <>, Positive range <>) of float;
type Square_Matrix (Size : Positive) is
 record
 Data : Matrix (1..Size, 1..Size);
 end record

```
- (a) Declare the variable `Small_Matrix` to be a  $4 \times 4$  square matrix and the variable `Large_Matrix` to be a  $120 \times 120$  square matrix.
  - (b) Write the package specification for a square matrix ADT declaring type `Square_Matrix` completely in the visible part of the package (this package should have no private part). Include operations to add two square matrices and to subtract two square matrices and an exception named `SIZE_ERROR` that is raised if an attempt is made to add or subtract square matrices of different sizes.
  - (c) Rewrite the package specification you did for part (b) making `Square_Matrix` a private type.
  - (d) What are the advantages and disadvantages of making `Square_Matrix` a private type?
  - (e) Using the matrix type defined in the package specification of part (b), declare three  $3 \times 3$  square matrix variables.
  - (f) Using the variables you declared in part (e) and the package specification of part (b), write a code fragment to read in two  $3 \times 3$  square matrices from the keyboard, add them together, and display the sum.
  - (g) Write the package body for the declaration you did for part (b). Adding and subtracting is accomplished by adding or subtracting the corresponding components in each matrix.



22. Read over programming problem 1 in the next section.

- (a) What are some advantages to declaring the complex type in the visible part of the package declaration rather than as a private type?
- (b) If the complex type were declared as a private type, what additional operations would you include in the package specification?

### Programming Problems

1. Write an ADT package that defines a type for complex numbers and functions for operators  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $<$ ,  $<=$ ,  $>$ , and  $>=$ . The type should *not* be a private type. The package should also include a function that returns the conjugate of a complex number. Also write a service package that provides input and output procedures for these complex numbers. Include Get and Put procedures that allow for interactive input and output and Get and Put procedures with a File parameter that allow for file input and output. Finally, write a driver program to test these two packages.

A complex number consists of two parts, a real part and an imaginary part. In algebra a complex number is represented as a quantity of two values often enclosed in parentheses like this

$$(5.27 + 17.34i)$$

where 5.27 is the real part of the complex number and 17.34 is the imaginary part. The  $i$  in this notation refers to the imaginary square root of -1. The conjugate of a complex number is obtained by simply changing the sign of the imaginary part of the number. For example, the conjugate of  $(5.27 + 17.34i)$  is  $(5.27 - 17.34i)$ .

Addition and subtraction of complex numbers is done by separately adding and subtracting the real and imaginary parts. For example,

$$\begin{aligned}(5.27 + 17.34i) + (6.11 - 5.30i) &= (11.38 + 12.04i) \\ (5.27 + 17.34i) - (6.11 - 5.30i) &= (-0.84 + 22.64i)\end{aligned}$$

Multiplication of two complex numbers may be done by applying the following formula:

$$(a + bi)(c + di) = (e + fi)$$

where

$$\begin{aligned}e &= ac - bd \\ f &= bc + ad\end{aligned}$$

For example,

$$(5.27 + 17.34i)(6.11 - 5.30i) = (124.10 + 78.02i)$$

The formula for division of two complex numbers is

$$(a + bi) / (c + di) = (e + fi)$$

where

$$\begin{aligned}e &= (ac + bd) / (c^2 + d^2) \\ f &= (bc - ad) / (c^2 + d^2)\end{aligned}$$

For example,

$$(5.27 + 17.34i) / (6.11 - 5.30i) = (-0.91 + 2.05i)$$

Two complex numbers are equal if, and only if, their real parts are equal and their imaginary parts are equal. Although complex numbers are not ordered like integer and real numbers, we can compare the magnitude of two complex numbers with the relational operators  $<$  and  $>$ . The magnitude of a complex number is calculated by taking the square root of the sum of the squares of its real part and its imaginary part. For example the magnitude of  $(5.27 + 17.34i)$  is 18.12 and the magnitude of  $(6.11 - 5.30i)$  is 8.09. Thus  $(5.27 + 17.34i)$  is greater than  $(6.11 - 5.30i)$ . In your package, use the square root function from the library that is supplied with your Ada compiler.

The form of the input and output values for the service I/O package should be the one used in our examples: two numbers separated by a sign enclosed in parentheses. The second number should have an  $i$  immediately after it. The Put procedure should have an Aft parameter to specify the number of decimal places desired for each component in the complex value. Use a default value of 2 for parameter Aft.

2. Rewrite your complex number package with the complex type declared as private. Make any necessary modifications and additions to the package specification. What ramification does making the type private have on the I/O service package?
3. Write a specification for a program that uses the complex number type to find the roots of a quadratic equation. Design, implement, and test a solution.
4. Specify, design, implement, and test an ADT package for fractions similar to the one discussed in Chapter 2. Explain why or why not you choose to make the fraction type private. Optional: Each student can give a specification to another student to design and implement. Then they can exchange again for testing and debugging the package.

## 4— Sets

### GOALS

After reading this chapter, you should be able to

- describe a set and its operations at the abstract level
- determine whether a set is an appropriate data structure for a specific problem
- use arrays of Boolean components to implement a set class whose elements are discrete objects
- design and implement generic packages
- design and implement generic subprograms
- understand designing for reuse

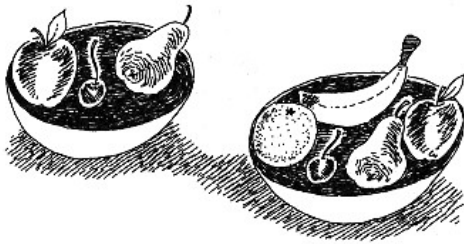
Chapter 3 presented the basic principles of designing data structures (collection classes) for data used in computer programs. In this chapter and the ones that follow, we use these principles to build a new range of data structures. These data structures are not predefined by most programming languages, as arrays and records are, but have to be created by the programmer before they can be used. Because these classes are fundamental to computer science, the Ada code for them can often be obtained from a public or private repository or purchased from vendors who market Ada components. But, as we said in an earlier chapter, our goal is to teach you how to develop software. You need to practice the fundamentals before you tackle more advanced software development.

We discuss each structure at three levels. First we *define* the abstract class: what does it "look" like and what are the logical operations on it? We write package specifications to specify the structure at this level. Once the package specification for the class is compiled, we can *use* it in applications. So next we look at one or more application examples that use the class. Finally we *build* one or more implementations of the class as package bodies.

In the second part of this chapter we show you how to write your own generic units. Until the appearance of modern programming languages like Ada, it was common for programmers to implement, verify, and test slightly different versions of a data structure for each application they developed. Even though there were only minor differences between the data structures in the different applications, programmers always "reinvented the wheel." Ada's generic units allow us to design and implement a *template* for a data structure. Then we simply supply some information to the Ada compiler, and it can use our template to generate a data structure customized specifically for our application.

### The Abstract Level

The abstract view of sets comes from discrete mathematics where a set is an *unordered* collection of items called set *members*. An item is either in the set (is a member) or not in the set (is not a member). A set often is written by enclosing the group of items in braces. Here, for example, is how we might write two different sets of fruit:



**Empty Set**

The set with no members.

**Universal Set**

The set containing all the values of a component type.

There are two special types of sets that are important: the **empty set**, which contains no values; and the **universal set**, a set that contains all the values of the base type. The universal set of fruit would contain all possible fruits. We use  $\{\}$  to indicate an empty set.

**Set Operations**

The most commonly used operation for sets is *membership testing*. Membership testing is simply asking whether an item is in the set. Here are some examples of membership testing:

| <i>Membership Test</i>                    | <i>Answer</i> |
|-------------------------------------------|---------------|
| Apple in the set { Apple Pear Cherry }    | True          |
| Kumquat in the set { Apple Banana Lemon } | False         |

*Union*, *intersection*, and *difference* are three operations that form a new set from two existing sets.

*Union* The union of two sets is a set consisting of those components that are in the first set *or* in the second set.

*Intersection* The intersection of two sets is a set consisting of those components occurring in the first set *and* in the second set.

*Difference* The difference between two sets is a set consisting of those components in the first set that are not in the second set.

Here are some examples of these three operations using sets of letters:

| <i>First Set</i> | <i>Second Set</i> | <i>Union</i> | <i>Intersection</i> | <i>Difference</i> |
|------------------|-------------------|--------------|---------------------|-------------------|
| {P Q R}          | {S}               | {P Q R S}    | {}                  | {P Q R}           |
| {S}              | {P Q R}           | {P Q R S}    | {}                  | {S}               |
| {A B C D}        | {B D F}           | {A B C D F}  | {B D}               | {A C}             |
| {L N}            | {L M N O P}       | {L M N O P}  | {L N}               | {}                |
| {}               | {X Y Z}           | {X Y Z}      | {}                  | {}                |
| {X Y Z}          | {}                | {X Y Z}      | {}                  | {X Y Z}           |

**Subset**

A set  $X$  is a subset of  $Y$  if each element of  $X$  is an element of  $Y$ . If at least one element of  $Y$  is not in  $X$ , then  $X$  is called a *proper* subset of  $Y$ . The set {Apple Pear Cherry} is a proper subset of the set {Orange Cherry Pear Peach Apple Banana}.

Another group of operations deals with subsets. A **subset** is a set contained within another set. Here are some examples of subset operations using sets of letters:

| <i>First Set</i> | <i>Second Set</i> | <i>First is subset of Second</i> | <i>First is proper subset of Second</i> |
|------------------|-------------------|----------------------------------|-----------------------------------------|
| {A E I O U}      | {A B C D E F}     | False                            | False                                   |
| {A B C D}        | {A B C}           | False                            | False                                   |
| {X Y Z}          | {X Y Z}           | True                             | False                                   |
| {X Y Z}          | {W X Y Z}         | True                             | True                                    |

Finally, we need some way of adding and removing individual elements from a set. We accomplish this task with operators that take a set and an element as operands. Here are some examples again using sets of letters:

| <i>Set</i>  | <i>Element</i> | <i>Addition of Element</i> | <i>Removal of Element</i> |
|-------------|----------------|----------------------------|---------------------------|
| {A E I O U} | B              | {A B E I O U}              | {A E I O U}               |
| {A B C D}   | B              | {A B C D}                  | {A C D}                   |

The addition operator is commonly used to create a set. Here, for example, is an algorithm for constructing a set of the letters from A to G:

```
Set := { }
for Letter in A .. G loop
 Set := Set + Letter
end loop
```

### *A Set Specification*

While some languages, like Pascal, have a predefined set type, Ada does not. We must create our own set class. Specification 4-1 is a specification for a set class whose members are the callable Bingo numbers (1–75) that we used in the last chapter.

#### **Specification 4-1— Bingo Number Set ADT**

```
with Bingo_Definitions;
use Bingo_Definitions;
package Bingo_Number_Set is

-- This package implements an object class for a set of Bingo Numbers.

 -- Set elements are Bingo numbers (define a synonym).
 subtype Element_Type is Bingo_Definitions.Callable_Number;

 -- The set class
 type Set_Type is private;

 -- A set object with no elements
 Empty_Set : constant Set_Type; -- A deferred constant (value is assigned in
 -- the private part of the package declaration)

 -- A set object with all possible elements
 Universal_Set : constant Set_Type; -- Another deferred constant

 function Is_Member (Element : in Element_Type;
 Set : in Set_Type) return Boolean;
 -- This function determines whether Element is a member of Set.

 function "+" (Left : in Set_Type; Right : in Set_Type) return Set_Type;
 -- Returns the union of the two sets
 -- (elements that are in either or both sets).
```

```

function "+" (Left : in Set_Type; Right : in Element_Type) return Set_Type;
-- Adds an element to a set
-- (the set is not changed if the element is already a member).

function "+" (Left : in Element_Type; Right : in Set_Type) return Set_Type;
-- Adds an element to a set
-- (the set is not changed if the element is already a member).

function "*" (Left : in Set_Type; Right : in Set_Type) return Set_Type;
-- Returns the intersection of the two sets
-- (elements occurring in both sets).

function "-" (Left : in Set_Type; Right : in Set_Type) return Set_Type;
-- Returns the difference of the two sets
-- (elements in Left and not in Right).

function "-" (Left : in Set_Type; Right : in Element_Type) return Set_Type;
-- Removes an element from the set
-- (the set is not changed if the element is not a member).

-- Relational Set Operators
-- The equality operators (= and /=) are
-- available for the private type Set_Type

function "<=" (Left : in Set_Type; Right : in Set_Type) return Boolean;
-- Returns True if Left is a subset of Right

function "<" (Left : in Set_Type; Right : in Set_Type) return Boolean;
-- Returns True if Left is a proper subset of Right

function ">=" (Left : in Set_Type; Right : in Set_Type) return Boolean;
-- Returns True if Right is a subset of Left

function ">" (Left : in Set_Type; Right : in Set_Type) return Boolean;
-- Returns True if Right is a proper subset of Left

private

 -- We fill this part in later.

end Bingo_Number_Set;

```

Specification 4-1 contains several Ada features that you may not be familiar with. First of all, Ada allows us to use operators like `+`, `or`, `*`, `rem`, and `not` as function names. In the specification of such a function, the operator symbol is enclosed in quotation marks. When a function name is an operator symbol, we can use it just like we use that operator. For example, we can use the function `"*"` from our set package to return the union of two sets. If `Set_1`, `Set_2`, and `Set_3` are all Bingo number sets, the following statement assigns the intersection of `Set_2` and `Set_3` to `Set_1`:

```
Set_1 := Set_2 * Set_3;
```

The actual parameters, `Set_2` and `Set_3`, in this call to function `"*"` are associated by position with the formal parameters `Left` and `Right`.

We can't use all the operators as function names. Ada does not permit us to use the short circuit operators (`and then` and `or else`) or the membership tests (`in` and `not in`) as function names. The equality operator (`=`) can only be used as a function name for comparing two limited private types.\* The inequality operator (`/=`) cannot be used as a function name. However, the Ada compiler creates a `"/=`" function for each `"="` function that you write.

---

#### Parameter Profile

The number of parameters, their base types, their order, and the base type of the function result. The Ada compiler uses parameter profiles to resolve calls to overloaded subprogram names.

---

A second Ada feature used in the set ADT package specification is *overloading*. We have used the same name for different operations. For example,  `"+"`  is used to designate a function to form the union of two sets and two functions that add a single element to a set. You have frequently used overloaded subprogram names; there are four different `Put` procedures defined in package `Text_IO` (see Appendix D). The Ada compiler can determine from the **parameter profile** which of these overloaded subprograms to use. Here are some example function calls. Make sure you can correctly match them to the appropriate function in the Bingo number set package specification.

```
Set_1 : Bingo_Number_Set.Set_Type;
Set_2 : Bingo_Number_Set.Set_Type;
Set_3 : Bingo_Number_Set.Set_Type;
Num : Bingo_Definitions.Callable_Number
.
.
.
Set_1 := Set_2 + Set_3; -- Calling first function "+"
Set_1 := Set_2 + Num; -- Calling second function "+"
Set_1 := Num + Set_2; -- Calling third function "+"
```

Like any other subprogram defined in a package, calls to these functions from outside of the package must be qualified or made directly visible with a `use` clause. To qualify a function that is designated by one of Ada's operators we must use the same syntax (the function name coming before the parameters) that we normally use to call a function. Here are the qualified versions of the three function calls we made above:

```
Set_1 := Bingo_Number_Set."+" (Set_2, Set_3);
Set_1 := Bingo_Number_Set."+" (Set_2, Num);
Set_1 := Bingo_Number_Set."+" (Num, Set_2);
```

---

\*This restriction is removed in Ada 95.



In most cases we prefer qualification because of the additional program documentation supplied. However, we use the `use` clause with packages containing "operator" functions so that we can call these functions without qualification. But we continue to qualify other identifiers with their package name as we did in the declarations of `Set_1`, `Set_2`, `Set_3`, and `Num`.

A third Ada feature used in the `Bingo_Number_Set` package specification is *deferred constants*. `Bingo_Number_Set` contains two deferred constants, `Empty_Set` and `Universal_Set`. These constants, whose types are `private`, are declared in the visible portion of the package. The actual value of a deferred constant is not given in the visible part; it is deferred to the private part. This means that package users can use the named constant (because it is declared in the visible part), but they may not access any of its details (because they are declared in the private part).

Here are some example expressions that use our `Bingo_Number_Set` operations:

| <i>Expression</i>              | <i>Result</i>                                                       |
|--------------------------------|---------------------------------------------------------------------|
| <code>Set_1 + Set_2</code>     | The union of <code>Set_1</code> and <code>Set_2</code>              |
| <code>Set_1 + Element</code>   | The addition of <code>Element</code> to <code>Set_1</code>          |
| <code>Set_1 * Set_2</code>     | The intersection of <code>Set_1</code> and <code>Set_2</code>       |
| <code>Set_1 - Set_2</code>     | The difference of <code>Set_1</code> and <code>Set_2</code>         |
| <code>Set_1 - Element</code>   | The removal of <code>Element</code> from <code>Set_1</code>         |
| <code>Set_1 = Set_2</code>     | True if the sets are identical                                      |
| <code>Set_1 /= Set_2</code>    | True if the sets are not identical                                  |
| <code>Set_1 &lt;= Set_2</code> | True if <code>Set_1</code> is a subset of <code>Set_2</code>        |
| <code>Set_1 &lt; Set_2</code>  | True if <code>Set_1</code> is a proper subset of <code>Set_2</code> |
| <code>Set_1 &gt;= Set_2</code> | True if <code>Set_2</code> is a subset of <code>Set_1</code>        |
| <code>Set_1 &gt; Set_2</code>  | True if <code>Set_2</code> is a proper subset of <code>Set_1</code> |

### The Application Level

In Chapter 3 we developed a Bingo game simulation program. We used a Bingo Basket object to model the random drawing of Bingo numbers. The basket object must ensure that a number is not called twice in one game—that is, the elements are unique. The numbers can be called in any order, so the collection can be considered unordered. A number has either been called or it hasn't been called. We now know a data class that has these properties and behaviors: a set.

#### *Bingo Number Selection*

We can implement the Bingo Basket as either a set of numbers that have not been called (those numbers that are still in the basket) or a set of numbers that already have been called (those numbers outside of the basket). Let's choose the first method (the second is left as an exercise). The specification of the Bingo Basket object (Specification 3–4, page 118) contains two operations: `Load` and `Draw`.

We can implement the `Load` procedure, which puts the numbers into the basket, by assigning the universal set to our basket set. Remember that the universal set is the set that contains all the values of the element type. Because Bingo numbers range from 1 to 75, the

universal set has 75 elements—each of the Bingo values. The `Draw` procedure requires that we select a number from this set at random. Random selection is not one of our set class operations. We could add such an operation to our set class, but this is not consistent with our mathematical view of this class. It is better for the Bingo number set object to have another object to supply it with a random number. Such objects, which are commonly used in simulation programs, are called *random number generators*. The implementation of a random number generator is beyond the scope of this text. We have, however, supplied a generic random number generator package on the disk that accompanies this text (listing in Appendix K).<sup>\*</sup> The operation we use returns a random number in the domain of `Bingo_Definitions.Callable_Number` between 1 and 75. After obtaining a random Bingo number, we use the membership operation to make sure that number has not already been drawn. If we haven't already drawn it, we remove it from the Bingo Basket set. Body 4-1 is the package body for our Bingo Basket object.

#### Body 4-1— Bingo Basket ADO

```
with Bingo_Number_Set; -- Set of Bingo numbers class
use Bingo_Number_Set; -- Make types and operations visible without qualification
with Discrete_Random; -- A generic random number generator
package body Bingo_Basket is

 -- Instantiate a random number generator for Bingo numbers.
 package Bingo_Generator is new Discrete_Random
 (Result_Subtype => Bingo_Definitions.Callable_Number);

 -- The set of numbers in the Bingo Basket
 The_Basket : Bingo_Number_Set.Set_Type;

 procedure Load is
 begin
 The_Basket := Bingo_Number_Set.Universal_Set;
 end Load;

 procedure Draw (Letter : out Bingo_Definitions.Bingo_Letter;
 Number : out Bingo_Definitions.Callable_Number) is

 The_Number : Bingo_Definitions.Callable_Number; -- A potential Bingo Number

 begin
 -- If the basket is empty raise the exception.
 if The_Basket = Bingo_Number_Set.Empty_Set then
 raise EMPTY;
 end if;
 -- Else get a number that is still in the Basket.
 loop
 -- "Choose" a number at random.
 The_Number := Bingo_Generator.Random;
```

---

<sup>\*</sup>Ada 95 includes several random number generators in package `Ada.Numerics`.

```

 -- If the number chosen is still in the basket, exit,
 -- otherwise go back and "choose" another number.
 exit when Is_Member (The_Number, The_Basket);
end loop;

-- Remove the number from the basket.
The_Basket := The_Basket - The_Number;
-- Copy number to the out parameter
Number := The_Number;

-- Determine the letter using the types in Bingo_Definitions.
case The_Number is
 when B_Range => Letter := B;
 when I_Range => Letter := I;
 when N_Range => Letter := N;
 when G_Range => Letter := G;
 when O_Range => Letter := O;
end case;
end Draw;
end Bingo_Basket;

```

The `Bingo_Basket` package uses the `Bingo_Number_Set` class without any idea of how it is implemented, just like you have used arrays before you knew how they were implemented. The set's specification, given earlier in this chapter, told us what the class is and how to use it but nothing about how it is implemented. In the next section we look at the details of implementing the `Bingo_Number_Set` class.

The algorithm used to implement the procedure `Draw` a number is not very efficient. Exercise 8 describes a more efficient algorithm and asks you to implement it.

### The Implementation Level

Now that you understand the set class and have seen a simple application that uses it, let's see how to implement it. Earlier we said that a set is an *unordered* collection of items. An item is either in the set (is a member) or not in the set (is not a member). We can use type `Boolean` to model this two-state behavior (True if the value is in the set, False if it's not). We need one Boolean value for *each* possible item in the set. For example, our set of callable Bingo numbers needs 75 Boolean values.

We can use an array of Boolean components to store all these values. Here are the declarations of such an array and a set object:

```
type Set_Type is array (Element_Type) of Boolean;
```

```
My_Set : Set_Type;
```

The array variable `My_Set` is shown in Figure 4-1. Subtype `Element_Type` was declared in the declaration of package `Bingo_Number_Set` as a synonym of `Bingo_Definitions.Callable_Number`. The indices of the array are the possible set items.

The value of a particular element in the array indicates whether or not that component is a member of the set. For example, the following code fragment displays whether or not 4 is a member of `My_Set`.

| My_Set |       |
|--------|-------|
| 1      | True  |
| 2      | False |
| 3      | True  |
| 4      | False |
| 5      | False |
|        | ⋮     |
|        | ⋮     |
| 73     | False |
| 74     | True  |
| 75     | False |

Figure 4-1  
A Set Stored as an  
Array of Booleans

```
if My_Set(4) then
 Text_IO.Put_Line ("4 is a member of the set");
else
 Text_IO.Put_Line ("4 is not a member of the set");
end if;
```

The empty set is an array in which all components are False and the universal set is one in which all components are True. We now can fill in the private part of package specification `Bingo_Number_Set`. This part must include the full declarations for type `Set_Type` and deferred constants `Empty_Set` and `Universal_Set`. We use array aggregates to assign values to the two constant arrays.

```
private -- Private part of Bingo Number Set package

 type Set_Type is array (Element_Type) of Boolean;

 -- Completion or deferred constants
 Empty_Set : constant Set_Type := (Element_Type => False);
 Universal_Set : constant Set_Type := (Element_Type => True);

end Bingo_Number_Set;
```

Because each possible element in the set is represented by a Boolean value, the implementation of most of the set operations in this package use the logical operators `and`, `or`, and `not`. Let's look at an example. Given the declarations

```
Set_One : Bingo_Number_Set.Set_Type; -- A set of Bingo_Numbers
Set_Two : Bingo_Number_Set.Set_Type; -- Another set of Bingo_Numbers
```

Figure 4-2 shows what the array structures look like if `Set_One` contains 1, 74, and 3 and `Set_Two` contains 4, 5, 74, 75, and 3.

|    | Set_One | Set_Two | Union | Intersection | Difference |
|----|---------|---------|-------|--------------|------------|
| 1  | True    | False   | True  | False        | True       |
| 2  | False   | False   | False | False        | False      |
| 3  | True    | True    | True  | True         | False      |
| 4  | False   | True    | True  | False        | False      |
| 5  | False   | True    | True  | False        | False      |
|    | .       | .       | .     | .            | .          |
|    | .       | .       | .     | .            | .          |
|    | .       | .       | .     | .            | .          |
| 73 | False   | False   | False | False        | False      |
| 74 | True    | True    | True  | True         | False      |
| 75 | False   | True    | True  | False        | False      |

Figure 4-2  
Examples of Set Operations

Note that the elements in the Union set are the result of *or*-ing the corresponding elements in the two sets, and the elements in the Intersection set are the result of *and*-ing the corresponding elements. The difference set is slightly more complicated. To be in the difference set, an element must be in `Set_One` but not in `Set_Two`. Thus each element in the difference array is calculated as

Difference Element := Set\_One Element and not (Set\_Two Element)

For each of these three operations, we can use a loop to go through the set arrays one element at a time. There is, however, a simpler method. Ada's `and`, `or`, and `not` operators may be applied to *entire* arrays of Boolean elements. Operating on the entire array rather than on individual elements eliminates the need for a loop.

The subset operations involve Boolean operations similar to the difference operation. However, the subset operations return a single True or False rather than another set. We can carry out this test with the difference operation. We "subtract" `Set_Two` from `Set_One`. Any elements in `Set_One` that are not in `Set_Two` are in the difference set. `Set_One` is *not* a subset of `Set_Two` if it contains an element that is not in `Set_Two`. If all of `Set_One`'s elements are also in `Set_Two` the difference yields an empty set. Thus our test for subset is a test on whether the difference yields an empty set.

Is a Subset := (Set\_One - Set\_Two) = Empty\_Set

Body 4-2 contains the Ada code for all of all these algorithms.

**Body 4-2—****Bingo Number Set ADT Implements Specification 4-1**

```

package body Bingo_Number_Set is

function Is_Member (Element : in Element_Type;
 Set : in Set_Type) return Boolean is
begin
 return Set(Element);
end Is_Member;

function "+" (Left : in Set_Type; Right : in Set_Type) return Set_Type is
begin
 return Left or Right;
end "+";

function "+" (Left : in Set_Type; Right : in Element_Type) return Set_Type is

 Result : Set_Type;

begin
 Result := Left;
 Result(Right) := True; -- Add the new element to the set.
 return Result;
end "+";

function "+" (Left : in Element_Type; Right : in Set_Type) return Set_Type is

 Result : Set_Type;

begin
 Result := Right;
 Result(Left) := True; -- Add the new element to the set.
 return Result;
end "+";

function "*" (Left : in Set_Type; Right : in Set_Type) return Set_Type is
begin
 return Left and Right;
end "*";

function "-" (Left : in Set_Type; Right : in Set_Type) return Set_Type is
begin
 return Left and not Right;
end "-";

```

```

function "-" (Left : in Set_Type; Right : in Element_Type) return Set_Type is

 Result : Set_Type;

begin
 Result := Left;
 Result(Right) := False; -- Remove the element from the set.
 return Result;
end "-";

function "<=" (Left : in Set_Type; Right : in Set_Type) return Boolean is

 Is_A_Subset : Boolean; -- Value to return

begin
 Is_A_Subset := (Left - Right) = Empty_Set;
 return Is_A_Subset;
end "<=";

function "<" (Left : in Set_Type; Right : in Set_Type) return Boolean is

 Result : Boolean; -- Value to return

begin
 if Left = Right then -- If the sets are equal, not a proper subset.
 Result := False;
 else
 Result := Left <= Right; -- If not equal, test for subset
 end if; -- using the function above.
 return Result;
end "<";

function ">=" (Left : in Set_Type; Right : in Set_Type) return Boolean is
begin
 return Right <= Left; -- Simply reverse the order of the parameters
 -- and use the operation we've already written.
end ">=";

function ">" (Left : in Set_Type; Right : in Set_Type) return Boolean is
begin
 return Right < Left; -- Simply reverse the order of the parameters
 -- and use the operation we've already written.
end ">";

end Bingo_Number_Set;

```

The set elements in our package are used as array indexes. Therefore, this implementation technique is limited to sets with discrete elements. We cannot, for example, use this

method for a set of `Float` numbers, a set of `Strings`, or a set of student records. In Chapters 8, 12, and 15 we develop data structures that can be used to implement sets with nondiscrete components.

### Analysis of the Set Algorithms

There are no loop statements in our set package body. The time necessary to carry out each operation does not depend on the number of items in a set. Therefore the Big-O for all operations in this implementation of the set class is  $O(1)$ .

Many of the set operations contain array expressions such as the expression `Left or Right` in the set union operation. This expression creates an array of Boolean components whose values are the result of *or*-ing the individual components in `Left` and `Right`. Even though many Ada compilers translate this expression into a single machine language instruction, the amount of time it takes to process an array is proportional to the size of the array. Similarly, the amount of time required for array assignment is also proportional to the size of the array. Because the array size is *constant for any set*, the amount of time for any operation on that set is also constant. Remember a *constant* time does not mean a *small* time. A constant time means that it always takes the *same* time.

### Programming for Reuse: Generic Units

Designing software modules for reuse is an important aspect of software engineering. The goal of software reuse is to make it possible for a software engineer to develop an application by combining various software modules much like today's hardware engineer develops new circuits by combining standard electronic components. Ada's generic units provide a powerful tool for designing reusable software components.

Looking for things that are familiar is an important problem-solving technique. A good programmer immediately recognizes a problem that he or she has solved before and plugs in the solution. For example, in the previous sections we developed a package to represent a set of Bingo numbers. If you were asked to write a program that required the use of a set of letters (A through Z), you would recognize a similarity with the Bingo number set. It would not take much effort on your part to modify the Bingo number set package to use in your new program.

---

#### Generic Unit

A template for a package or subprogram.

#### Instance

A package or subprogram created from a generic unit.

---

Ada's **generic units** make reusing previous solutions even easier. We write one general solution to the problem (the generic unit), then we let the compiler modify the general solution to satisfy each particular problem (see Figure 4-3). The specific solution created by the compiler is called an **instance** of our generic solution.

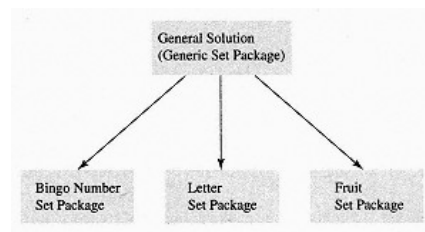


Figure 4-3  
The Ada Compiler Can Produce Specific Packages  
(Instances) from a Single Generic Package



Generic units allow us to save more than the time and effort it takes to use an editor to modify the solution. Because the Ada compiler makes the modifications without error, we usually need only test the solution one time. For example, if our generic package works for a set of Bingo numbers, it should also work for a set of letters. We do not have to test the set portion of our new program. We can concentrate our testing efforts on the portions of the problem that are new. Because the generic units are kept in our Ada library, we can easily use them in other programs or share them with other programmers.

---

**Generic Formal Parameter**

A parameter defined in a generic unit declaration. Used to customize a generic unit for a specific problem.

**Instantiation**

A declaration that creates an instance of a generic unit.

---

But how does the compiler know how to make the specific version of the generic package that we need? **Generic formal parameters** make generic packages flexible so that they can be used in a variety of problems. These parameters give the Ada compiler the information it needs to produce a unit customized for a particular problem from the generic unit. Perhaps without knowing it, you already have used generic parameters. In each **instantiation** of a package for the input and output of enumeration or numeric types you use a generic parameter.

Here is a code fragment to illustrate the use of generic parameters. The instantiation creates a package for the input and output of values of type `Color_Type`.

```
type Color_Type is (Red, Green, Blue, Magenta, Cyan, Yellow);

package Color_IO is new Text_IO Enumeration_IO (Enum => Color_Type);
```

`Enum` is a generic *formal* parameter of the generic package `Enumeration_IO` for which we have supplied an *actual* value of `Color_Type`. This generic parameter is different than the parameters we use with subprograms; it is a *type*, not a value. When this *instantiation* is elaborated, the Ada compiler constructs a package called `Color_IO` from the generic package `Enumeration_IO` by substituting `Color_Type` for `Enum`. It is as though you had a copy of `Enumeration_IO` and used your program editor to change every occurrence of `Enum` to `Color_Type`.

Let's see how we can write a generic set package that we then can use anytime we need a set class in a program. We base this generic package on the Bingo number set package developed earlier in this chapter. Like any package, a generic package consists of two parts: the package specification and the package body. Here is a simplified EBNF definition of a generic package specification:

```
generic_package_declaration ::= generic
 {generic_parameter_declaration}
 package_specification;
```

You can see that the generic formal parameters are declared between the reserved word `generic` and the package specification. The definition of package specification is exactly the same as with ordinary packages.

The specification for a generic set package is shown in Specification 4-2. Compare this declaration to that of the Bingo number set package (Specification 4-1) on page 132.

#### Specification 4-2— Discrete Set ADT

```
generic

 type Element_Type is (<>);
```

```

package Discrete_Set is

-- This package implements an ADT for a set of discrete elements.

 -- The Set Class
type Set_Type is private;

 -- A set with no elements
Empty_Set : constant Set_Type;

 -- A set with all possible elements
Universal_Set : constant Set_Type;

function Is_Member (Element : in Element_Type;
 Set : in Set_Type) return Boolean;
-- This function determines whether Element is a member of Set.

function "+" (Left : in Set_Type; Right : in Set_Type) return Set_Type;
-- Returns the union of the two sets
-- (elements that are in either or both sets).

function "+" (Left : in Set_Type; Right : in Element_Type) return Set_Type;
-- Adds an element to a set
-- (the set is not changed if the element is already a member).

function "+" (Left : in Element_Type; Right : in Set_Type) return Set_Type;
-- Adds an element to a set
-- (the set is not changed if the element is already a member).

function "*" (Left : in Set_Type; Right : in Set_Type) return Set_Type;
-- Returns the intersection of the two sets
-- (elements occurring in both sets).

function "-" (Left : in Set_Type; Right : in Set_Type) return Set_Type;
-- Returns the difference of the two sets
-- (elements in Left and not in Right).

function "-" (Left : in Set_Type; Right : in Element_Type) return Set_Type;
-- Removes an element from the set
-- (the set is not changed if the element is not a member).

--
-- Relational Set Operators
-- The equality operators (= and /=) are
-- available for the private type Set_Type.

```

```

function "<=" (Left : in Set_Type; Right : in Set_Type) return Boolean;
-- Returns True if Left is a subset of Right.

function "<" (Left : in Set_Type; Right : in Set_Type) return Boolean;
-- Returns True if Left is a proper subset of Right.

function ">=" (Left : in Set_Type; Right : in Set_Type) return Boolean;
-- Returns True if Right is a subset of Left.

function ">" (Left : in Set_Type; Right : in Set_Type) return Boolean;
-- Returns True if Right is a proper subset of Left.

```

```
private
```

```
type Set_Type is array (Element_Type) of Boolean;
```

```

-- Completion of deferred constants
Empty_Set : constant Set_Type := (Element_Type => False);
Universal_Set : constant Set_Type := (Element_Type => True);

```

```
end Discrete_Set;
```

The only differences between Specifications 4-1 and 4-2 are the package names and the definitions of `Element_Type`. In package `Discrete_Set`, `Element_Type` is a generic formal parameter. Like any parameters, generic formal parameters are associated with actual parameters. This association is made when a package is instantiated from a generic package. For example, the following code fragment creates a set package for lowercase letters, a set package for fruit, and a set package for Bingo numbers:

```

subtype Letters is Character range 'a'.. 'z';
type Fruit_Type is (Apple, Banana, Blueberry, Cherry, Plum, Strawberry);

package Letter_Set is new Discrete_Set (Element_Type => Letters);
package Fruit_Set is new Discrete_Set (Element_Type => Fruit_Type);
package Bingo_Set is new Discrete_Set
 (Element_Type => Bingo_Definitions.Callable_Numbers);

```

Ada creates the package `Letter_Set` from the generic package `Discrete_Set` by using the generic actual parameter `Letters` in place of the generic formal parameter `Element_Type`. Similarly, Ada creates the package `Fruit_Set` by using the generic actual parameter `Fruit_Type` in place of the generic formal parameter `Element_Type`. Finally it creates the package `Bingo_Set` by using the generic actual parameter `Callable_Numbers` in place of the generic formal parameter `Element_Type`.

Ada also allows us to write generic subprograms. These generic units make it easy to reuse a single procedure or function. Like generic packages, generic subprograms may have generic formal parameters that are matched with actual parameters when a subprogram is instantiated from the generic template. Here is a more complete set of EBNF definitions for the declaration of generic units including both packages and subprograms:

```

generic_declaration ::= generic_specification

generic_specification ::= generic_formal_part
 subprogram_specification |
 generic_formal_part
 package_specification

generic_formal_part ::= generic {generic_parameter_declaration}

generic_parameter_declaration ::= type identifier is generic_type_definition; |
 private_type_definition |
 with subprogram_specification; |
 identifier : type_mark [:= expression];

generic_type_definition ::= (<>) |
 range <> |
 digits <> |
 array_type_definition

private_type_definition ::=

 type identifier [discriminant_part] is [limited] private;

```

This set of EBNF definitions includes the four kinds of generic parameters. Just as there are rules for subprogram formal and actual parameters, there are rules for generic formal and actual parameters. We discuss the first three kinds of generic parameters (types, private types, and subprograms) below. We introduce generic formal objects in Chapter 5.

### ***Generic Formal Types***

Generic formal types allow us to choose at instantiation time the types we want to use in the new package. The generic formal parameter `Element_Type` in our discrete set package (Specification 4-2) is an example of a generic formal type. The generic formal parameter `Enum` in the generic package `Enumeration_IO` is another example.

Our simplified set of EBNF definitions includes four different generic type definitions. Each generic type definitions specifies

1. what the rules are for matching the actual and formal types
2. how the type may be used in the package

We discuss each of the four generic type definitions in terms of these two criteria.

### **Discrete Types**

Here is the declaration of `Discrete_Set`'s generic formal parameter.

```
type Element_Type is (<>);
```

The `(<>)` in this declaration means that any *discrete* type may be used as an actual parameter in an instantiation made from this generic package.\* Thus we can instantiate set

---

\*We also use the symbol `<>` (called "box") in the declaration of unconstrained array types.

packages for integer types (like Bingo numbers), character types (like lowercase letters), and enumeration types (like `Fruit_Type`). However, we cannot use types `Float`, `String`, or any array type as a generic actual parameter with our generic set package.

Why did we choose to restrict our set elements to discrete types? In our design of the original Bingo number set package, our algorithms were based on using set elements as array indices. The type of the set elements must therefore be a discrete type. We use `(<>)` in the declaration of the generic formal parameter `Element_Type` to restrict actual parameters to discrete types so that the same algorithms could be used.

In addition to restricting the types of generic actual parameters that can be used, the generic type definition also restricts how the type may be used in the generic package body. For example, the operations permitted in package `Discrete_Set` on objects of type `Element_Type` are restricted to those available to *all* discrete types. Thus we may use the assignment operator `(:=)`, the relational operators `(>)`, `>=`, `<`, `<=`, `=`, and `/=`, and the discrete attributes `'First`, `'Last`, `'Pos`, `'Val`, `'Pred`, and `'Succ` with such objects. We *cannot* use any arithmetic operators. Arithmetic operators are available for some discrete types (integer types and subtypes), not *all* discrete types.

Body 4-3 is the complete package body for our generic set package.

#### Body 4-3—

#### Discrete Set ADT Implements Specification 4-2

```
package body Discrete_Set is

function Is_Member (Element : in Element_Type;
 Set : in Set_Type) return Boolean is
begin
 return Set (Element);
end Is_Member;

function "+" (Left : in Set_Type; Right : in Set_Type) return Set_Type is
begin
 return Left or Right;
end "+";

function "+" (Left : in Set_Type; Right : in Element_Type) return Set_Type is

 Result : Set_Type;

begin
 Result := Left;
 Result(Right) := True; -- Add the new element to the set.
 return Result;
end "+";
```

```

function "+" (Left : in Element_Type; Right : in Set_Type) return Set_Type is

 Result : Set_Type;

begin
 Result := Right;
 Result(Left) := True; -- Add the new element to the set.
 return Result;
end "+";

function "*" (Left : in Set_Type; Right : in Set_Type) return Set_Type is
begin
 return Left and Right;
end "*";

function "-" (Left : in Set_Type; Right : in Set_Type) return Set_Type is
begin
 return Left and not Right;
end "-";

function "-" (Left : in Set_Type; Right : in Element_Type) return Set_Type is

 Result : Set_Type;

begin
 Result := Left;
 Result(Right) := False; -- Remove the element from the set.
 return Result;
end "-";

function "<=" (Left : in Set_Type; Right : in Set_Type) return Boolean is

 Is_A_Subset : Boolean; -- Value to return

begin
 Is_A_Subset := (Left - Right) = Empty_Set;
 return Is_A_Subset;
end "<=";

function "<"
 (Left : in Set_Type; Right : in Set_Type) return Boolean is

 Result : Boolean; -- Value to return

begin
 if Left = Right then -- If the sets are equal, not a proper subset.
 Result := False;
 end if;
end "<";

```

```

 else
 Result := Left <= Right; -- If not equal, test for subset
 end if;
 return Result;
end "<";

function ">=" (Left : in Set_Type; Right : in Set_Type) return Boolean is
begin
 return Right <= Left; -- Simply reverse the order of the parameters.
 -- and use the operation we've already written.
end ">=";

function ">" (Left : in Set_Type; Right : in Set_Type) return Boolean is
begin
 return Right < Left; -- Simply reverse the order of the parameters
 -- and use the operation we've already written.
end ">";

end Discrete_Set;

```

## Integer Types

A formal generic type parameter defined with `range <>` matches an actual parameter of any integer type or integer subtype. The operations that can be used in the package for this type include all of the integer operations with which you are familiar. These include assignment, the arithmetic operators, the relational operators, and the attributes defined for integer types (including 'First, 'Last, 'Succ, and 'Pred).

Let's look at an example. Here's the declaration of a very simple service package that contains three operations for integer values: a function for determining the smaller of two values, a function for determining the larger of two values, and a procedure for swapping the values of two integer variables.

```

generic

 type Whole_Number_Type is range <>; -- Generic formal type parameter

package Integer_Ops is

 function Maximum (Left : in Whole_Number_Type;
 Right : in Whole_Number_Type) return
 Whole_Number_Type;

 function Minimum (Left : in Whole_Number_Type;
 Right : in Whole_Number_Type) return
 Whole_Number_Type;

 procedure Swap (Left : in out Whole_Number_Type;
 Right : in out Whole_Number_Type);

end Integer_Ops;

```

Here is the body of this generic package.

```

package body Integer_Ops is

function Maximum (Left : in Whole_Number_Type;
 Right : in Whole_Number_Type) return
 Whole_Number_Type is
begin
 if Left > Right then
 return Left;
 else
 return Right;
 end if;
end Maximum;

function Minimum (Left : in Whole_Number_Type;
 Right : in Whole_Number_Type) return
 Whole_Number_Type is
begin
 if Left < Right then
 return Left;
 else
 return Right;
 end if;
end Minimum;

procedure Swap (Left : in out Whole_Number_Type;
 Right : in out Whole_Number_Type) is
 Temp : Whole_Number_Type;
begin
 Temp := Left;
 Left := Right;
 Right := Temp;
end Swap;

end Integer_Ops;

```

And here are some examples of instantiations of packages from this generic package:

```

subtype Quiz_Score_Type is Integer range 1..10;
type Feet is range 1..25;

package Quiz_Ops is new Integer_Ops
 (Whole_Number_Type => Quiz_Score_Type);
package Feet_Ops is new Integer_Ops (Whole_Number_Type => Feet);

```

When the package Quiz\_Ops is instantiated, at every place that Whole\_Number\_Type is used in the generic template Integer\_Ops it is replaced with the actual type Quiz\_Score\_Type. Similarly, when the package Feet\_Ops is instantiated, Whole\_Number\_Type is replaced with the actual type Feet.



## Float Types

A formal generic type parameter defined with `digits <>` matches an actual parameter of any float type or float subtype. The operations that can be used in the package for this type include all of the float operations with which you are familiar. These include assignment, the arithmetic operators, the relational operators, and the attributes defined for float types (including `'First`, `'Last`, and `'Digits`).

Here is a code fragment of a generic package declaration with a generic formal float type parameter:

```
generic

 type Number_Type is digits <>; -- Generic formal type parameter

package Number_Ops is
.
.
.
```

We can use this generic package to instantiate packages for any float type as illustrated by these examples.

```
type Pounds is digits 8 range 0.0..1000.0;
type Ounces is digits 5 range 0.0..16.0;

package Pound_Ops is new Number_Ops (Number_Type => Pounds);
package Ounces_Ops is new Number_Ops (Number_Type => Ounces);
```

## Array Types

The declarations of generic formal array types look just like the declarations that we have used for ordinary array types. Usually, the declaration of a generic formal array type makes use of previously declared generic formal types, as in this example:

```
generic

 type Discrete_Type is (<>);
 type Int_Type is range <>;
 type Int_Array is array (Discrete_Type) of Int_Type;
 type Discrete_Array is array (Int_Type range <>) of Discrete_Type;
 type Boolean_Array is array (Natural range <>) of Boolean;
 type Char_Array is array (Int_Type) of Character;

package Array_Example is
.
.
.
```

In this example, the formal array type `Int_Array` is declared as a constrained array whose index type and bounds are defined by the formal type parameter `Discrete_Type` and whose component type is defined by the formal type parameter `Int_Type`. The formal array type `Discrete_Array` is declared as an unconstrained array whose index type is defined by the formal type parameter `Int_Type` and whose component type is defined by the formal type parameter `Discrete_Type`. The formal array type `Boolean_Array`

is declared as an unconstrained array whose index type is `Natural` and whose components are Boolean values. Finally, the formal array type `Char_Array` is declared as a constrained array whose index type and bounds are defined by the formal type parameter `Int_Type` and whose components are character values.

The rules for matching actual and formal array types involve five conditions.

1. The actual array type and the formal array type must be either both constrained or both unconstrained.
2. The index type of the actual array type and the formal array type must have the same base type.
3. If the array parameter is constrained, the lower and upper bounds of the actual array type and the formal array parameter must be the same.
4. The component type must be the same for the actual array type as for the formal array type.
5. The actual array type and the formal array type must have the same number of dimensions.

Let's look at an example instantiation of generic package `Array_Example`.

```
subtype Small_Int is Integer range 1..10;

-- Constrained array type
type Small_Array is array (Boolean) of Small_Int;

-- Unconstrained array type
type Truth_Array is array (Small_Int range <>) of Boolean;

-- Constrained array type
subtype Small_String is String (Small_Int);

package Example is new Array_Example (Discrete_Type => Boolean,
 Int_Type => Small_Int,
 Int_Array => Small_Array,
 Discrete_Array => Truth_Array,
 Boolean_Array => Truth_Array,
 Char_Array => Small_String);
```

For each of the array parameters in this example, go through the five conditions that must be met for a match between an actual and a formal array parameter.

The operations that we can use in the package for formal array types are the operations available for any array type, including assignment, indexing, slicing, the relational operators (for one-dimensional arrays whose components are discrete types), and the array attributes (including `'First`, `'Last`, `'Length`, and `'Range`).

Let's look at another example. The following generic package uses most of the generic formal type parameters that we have seen. Here is the generic package declaration:

```
generic

type Component_Type is digits <>;
type Index_Type is (<>);
type Float_Array is array (Index_Type range <>) of Component_Type;
```

package Float\_Array\_Ops is

```

function "*" (Left : in Float_Array; Right : in Component_Type)
 return Float_Array;
-- This function returns an array of float components equal to the
-- array parameter Left with each component multiplied by Right.

function "/" (Left : in Float_Array; Right : in Component_Type)
 return Float_Array;
-- This function returns an array of float components equal to the
-- array parameter Left with each component divided by Right.

function "+" (Left : in Float_Array; Right : in Component_Type)
 return Float_Array;
-- This function returns an array of float components equal to the
-- array parameter Left with Right added to each component.

function "-" (Left : in Float_Array; Right : in Component_Type)
 return Float_Array;
-- This function returns an array of float components equal to the
-- array parameter Left with Right subtracted from each component.

```

end Float\_Array\_Ops;

Here is the body of generic package Float\_Array\_Ops:

package body Float\_Array\_Ops is

```

function "*" (Left : in Float_Array; Right : in Component_Type)
 return Float_Array is

 subtype Local_Array is Float_Array (Left'Range); -- Type for Result

 Result : Local_Array; -- The Result array

begin
 for Index in Left'Range loop
 Result (Index) := Left (Index) * Right;
 end loop;
 return Result;
end "*";

```

```

function "/" (Left : in Float_Array; Right : in Component_Type)
 return Float_Array is

 subtype Local_Array is Float_Array (Left'Range); -- Type for Result

 Result : Local_Array; -- The Result array

begin

```

```

 for Index in Left'Range loop
 Result (Index) := Left (Index) / Right;
 end loop;
 return Result;
end "/";

function "+" (Left : in Float_Array; Right : in Component_Type)
 return Float_Array is

 subtype Local_Array is Float_Array (Left'Range); -- Type for Result

 Result : Local_Array; -- The Result array

begin
 for Index in Left'Range loop
 Result (Index) := Left (Index) + Right;
 end loop;
 return Result;
end "+";

function "-" (Left : in Float_Array; Right : in Component_Type)
 return Float_Array is

 subtype Local_Array is Float_Array (Left'Range); -- Type for Result

 Result : Local_Array; -- The Result array

begin
 for Index in Left'Range loop
 Result (Index) := Left (Index) - Right;
 end loop;
 return Result;
end "-";

end Float_Array_Ops;

```

### ***Generic Formal Private Types***

A formal generic type parameter defined with `private` matches an actual parameter of *any* type. Because the parameter can be of any type, the operations are restricted to those that are common to *all* types. Only assignment (`:=`) and equality testing (`=` and `/=`) are available for all types, so these are the only operations permitted with a generic formal private type. Let's look at an example. Here is the declaration of a generic procedure that swaps the values of its two parameters:

```

generic

 type Swap_Type is private; -- Matches any type

procedure Swap_Values (Left : in out Swap_Type;
 Right : in out Swap_Type);

```

We can instantiate a swap procedure for any data type from this generic procedure. The following instantiations create procedures for swapping the values of two integer variables, two float variables, two character variables, and two student record variables. As with any overloaded procedure call, the Ada compiler uses the type of the actual parameters to determine which of our newly created `Swap` procedures to call.

```
procedure Swap is new Swap_Values (Swap_Type => Integer);
procedure Swap is new Swap_Values (Swap_Type => Float);
procedure Swap is new Swap_Values (Swap_Type => Character);
procedure Swap is new Swap_Values (Swap_Type => Student_Rec);
```

Now let's look at the body of the generic procedure `Swap_Values`. The only operation we need to swap two values is assignment.

```
procedure Swap_Values (Left : in out Swap_Type;
 Right : in out Swap_Type) is
 Temp : Swap_Type;
begin
 Temp := Left;
 Left := Right;
 Right:= Temp;
end Swap_Values;
```

We can also define a generic formal parameter as `limited private`. The addition of the reserved word `limited` limits our use of the type even further. We cannot use any operations with values of a `limited private` type. In the next section we show how we can supply operations that the generic unit may use with `private` or `limited private` types.

### ***Generic Formal Subprograms***

As we saw in the last section, generic formal private types allow us to instantiate generic units for any type. This generality comes with a price—the operations available for values of these types are very restricted. But, we can use generic subprogram parameters to obtain additional operations for `private` and `limited private` types. The actual subprograms are supplied by the programmer instantiating the generic unit. Let's look at an example. The following procedure sorts an array of values. These values can be any type.

```
generic

 type Element_Type is private;
 type Array_Type is array (Positive range <>) of Element_Type;
 with procedure Swap (First : in out Element_Type;
 Second : in out Element_Type);
 with function "<" (Left : in Element_Type;
 Right : in Element_Type) return Boolean;

procedure Selection_Sort (The_Array : in out Array_Type);
```

We must supply four actual parameters when we instantiate a sort procedure from this generic procedure: the array component type, the array type, a procedure for swapping two

components, and a function for comparing two components. Let's look at an example. Here are the necessary declarations to instantiate a procedure that sorts an array of inventory records by part number:

```

type Part_Rec is
 record
 Part_Number : Positive;
 Quantity : Natural;
 end record;

type Part_Array is array (Positive range <>) of Part_Rec;

-- A function to compare two inventory records.
function "<" (Left : in Part_Rec;
 Right : in Part_Rec) return Boolean is
begin
 return Left.Part_Number < Right.Part_Number;
end "<";

-- A procedure to swap two inventory records.
procedure Part_Swap is new Swap_Values (Swap_Type => Part_Rec);

-- The Sort procedure
procedure Part_Sort is new Selection_Sort (Element_Type => Part_Rec,
 Array_Type => Part_Array,
 Swap => Part_Swap,
 "<" => "<");

```

The instantiation of procedure `Part_Sort` requires two subprograms. We wrote the function for comparing two part records. We did not have to write a procedure to swap two part records; we instantiated the necessary procedure from the generic one we wrote earlier. Here is the body of our generic sort procedure:

```

procedure Selection_Sort (The_Array : in out Array_Type) is
 Min_Index : Positive;
begin
 for Pass_Count in The_Array'First .. The_Array'Last - 1 loop
 Min_Index := Pass_Count;
 for Place_Count in Pass_Count + 1 .. The_Array'Last loop
 if The_Array(Place_Count) < The_Array(Min_Index) then
 Min_Index := Place_Count;
 end if;
 end loop;
 if Min_Index /= Pass_Count then
 Swap (First => The_Array(Min_Index),
 Second => The_Array(Pass_Count));
 end if;
 end loop;
end Selection_Sort;

```

This body uses the "<" function supplied at instantiation to compare two array elements. It uses the `Swap` procedure supplied at instantiation to swap two array elements.

We began this last example of generic unit with a declaration of the unit. When developing a generic unit with generic formal private types, we usually do not know all of the operations needed by the body ahead of time. Therefore we must often develop the body and declaration simultaneously. The following case study illustrates this approach to design.

### **Case Study:**

#### ***A Reusable Binary Search Procedure***

It is important when developing a generic package or generic subprogram that we design it as generally as possible so it can be used in as many different applications as possible. In Chapter 2 we used a binary search procedure to illustrate testing. The binary search is useful in many applications. We can save a good deal of time in the future by developing a generic binary search procedure that can be instantiated for a wide variety of applications. First, let's look at the final version of the procedure from Chapter 2.

```
-- Final version from Chapter 2
procedure Binary_Search (List : in Array_Type;
 Value : in Element_Type;
 Location : out Natural) is
-- Search List for Value

-- Preconditions: List is in ascending order

-- Postconditions: Value is in List and List(Location) = Value or
-- Value is not in List and Location = 0

 Found : Boolean; -- Has Value been found yet?
 Midpoint : Natural; -- Index of search area's midpoint
 First_Index : Natural; -- First index in current search area
 Last_Index : Natural; -- Last index in current search area

begin -- Binary_Search
 -- Initialize
 Found := False;
 First_Index := 1;
 Last_Index := List'Last;

 -- Search until element found or the current search area is empty.
 -- Current search area is List (First_Index..Last_Index).

 while (First_Index <= Last_Index) and not Found loop
 -- Find middle element in the current search area.
 Midpoint := (First_Index + Last_Index) / 2;
 -- Compare Value to middle element in search area.
 if List(Midpoint) = Value then
 Found := True;
 elsif Value < List(Midpoint) then
 Last_Index := Midpoint - 1; -- Search 1st half
 else
 First_Index := Midpoint + 1; -- Search 2nd half
 end if;
 end loop;
end;
```

```

end loop;
-- Set value of Location.
if Found then
 Location := Midpoint;
else
 Location := 0;
end if;
end Binary_Search;

```

The types `Array_Type` and `Element_Type` used for this procedure were defined in the program `Test_Driver` in which this procedure was nested. To be general, we must be able to compile our procedure outside of any program—it must be a library unit (see Appendix J). We can accomplish this goal by making these two types generic formal parameters in a generic procedure declaration like this:

```

generic

 type Element_Type is _____;
 type Array_Type is _____;

procedure Binary_Search (List : in Array_Type;
 Value : in Element_Type;
 Location : out Natural);
-- Search List for Value

-- Preconditions: List is in ascending order

-- Postconditions: Value is in List and List(Location) = Value or
-- Value is not in List and Location = 0

```

What do we choose to fill in the blanks? We start by choosing `range <>` for type `Element_Type`. This choice restricts our array components to integer types. Type `Array_Type` must be an array whose components are type `Element_Type`. Let's declare our array type as an array with 100 integer components. Here is the resulting generic procedure declaration:

```

generic -- Version 1

 type Element_Type is range <>;
 type Array_Type is array (1..100) of Element_Type;

procedure Binary_Search (List : in Array_Type;
 Value : in Element_Type;
 Location : out Natural);

```

### Generalizing the Index

The declaration of Version 1 is still very restrictive. It only allows us to instantiate binary search procedures for searching arrays whose index range is 1..100 and whose components are some integer type. However, there are two things we can



do to make the array index range more general. First, we change the array type from a constrained array type to an unconstrained array type like this:

```
generic -- Version 2

 type Element_Type is range <>;
 type Array_Type is array (Integer range <>) of Element_Type;

procedure Binary_Search (List : in Array_Type;
 Value : in Element_Type;
 Location : out Natural);
```

Second, we need to make the type of the array index more general than `Integer`. Remember that we can use any discrete type for array index. To make our generic binary search procedure as general as possible, we need to allow the array index to be any discrete type. To do so we add a third generic parameter—the type of the array index. Now our generic procedure declaration looks like this:

```
generic -- Version 3

 type Element_Type is range <>;
 type Index_Type is (<>);
 type Array_Type is array (Index_Type range <>) of Element_Type;

procedure Binary_Search (List : in Array_Type;
 Value : in Element_Type;
 Location : out Natural);
```

The out parameter `Location` of procedure `Binary_Search` is used to return the index of the array component we are searching for. However, we have generalized the array type so that *any* discrete type may be used for an index. We must, therefore, change the type of `Location` to `Index_Type`. But now we have introduced another problem. The original procedure indicates that a search is unsuccessful by returning a value of 0 in `Location`. Obviously we can't return a 0 if our index type is some non-integer discrete (such as `Character`). The original design of our procedure's interface (parameters, preconditions, and postconditions) is inadequate for a more general implementation; our original procedure was not designed for reuse. Here is a more general interface:

```
generic -- Version 4

 type Element_Type is range <>;
 type Index_Type is (<>);
 type Array_Type is array (Index_Type range <>) of Element_Type;

procedure Binary_Search (List : in Array_Type;
 Value : in Element_Type;
 Success : out Boolean;
 Location : out Index_Type);

-- Search List for Value

-- Preconditions: List is in ascending order
```

```
-- Postconditions: Value is in List and Success and
-- List(Location) = Value
-- or
-- Value is not in List and not Success and
-- Location = Index_Type'Last.
```

We added the parameter `Success`. If `Value` is found in `List`, `Success` is set to `True`; otherwise `Success` is set to `False`. Here are some sample instantiations of this generic procedure:

```
type Month_Type is (January, February, March, April,
 May, June, July, August, September,
 October, November, December);

type Nat_Array is (Month_Type range <>) of Natural;
type Int_Array is (Positive range <>) of Integer;

procedure Nat_Search is new Binary_Search (Element_Type => Natural,
 Index_Type => Month_Type,
 Array_Type => Nat_Array);

procedure Int_Search is new Binary_Search (Element_Type => Integer,
 Index_Type => Positive,
 Array_Type => Int_Array);
```

The first instantiation creates a binary search procedure that searches for a particular natural number in an array whose index type is an enumeration type (`Month_Type`). The second instantiation creates a binary search procedure that searches for a particular whole number in an array indexed by a positive integer type. Figure 4-4 illustrates two arrays that could be searched by these two procedures.

(a) An array that procedure `Nat_Search` could search

| March | April | May | June | July | August |
|-------|-------|-----|------|------|--------|
| 0     | 7     | 14  | 19   | 27   | 193    |

(b) An array that procedure `Int_Search` could search

| 7   | 8   | 9   | 10 | 11 | 12 | 13  | 14  | 15  | 16  | 17  |
|-----|-----|-----|----|----|----|-----|-----|-----|-----|-----|
| -42 | -27 | -10 | 0  | 18 | 27 | 182 | 274 | 295 | 507 | 698 |

Figure 4-4  
Ordered Arrays for Binary Searching

Now let's turn to the body of this generic procedure. Obviously, the body on page 157 must have its parameters and postconditions changed to match our more general declaration. There are six other changes that we must make.

1. We must change the types of the local variables `Midpoint`, `First_Index`, and `Last_Index` from `Natural` to `Index_Type`.
2. We must change the initial value assigned to `First_Index` from `1` to `List 'First`.

3. We must calculate `Midpoint` in a manner suitable for *any* discrete type. When our array indexes were integers, we easily calculated the midpoint through simple algebra.

```
Midpoint := (First_Index + Last_Index) / 2;
```

Since the array index in our generic version may be any discrete type, we cannot use arithmetic operators directly. We must make use of the attributes `'Pos` and `'Val` to determine the midpoint.

```
Midpoint := Index_Type'Val ((Index_Type'Pos (First_Index) +
 Index_Type'Pos (Last_Index)) / 2);
```

4. Instead of subtracting or adding one to `Midpoint` to divide the current search area in half we must use `'Succ` and `'Pred`. We replace the lines

```
elsif Value < List(Midpoint) then
 Last_Index := Midpoint - 1; -- Search 1st half
else
 First_Index := Midpoint + 1; -- Search 2nd half
```

with

```
elsif Value < List (Midpoint) then
 Last_Index := Index_Type'Pred (Midpoint); -- Search 1st half
else
 First_Index := Index_Type'Succ (Midpoint); -- Search 2nd half
```

5. If `Midpoint` is equal to `List 'First` or to `List 'Last`, we may attempt to find the successor of the last value of the type or the predecessor of the first value thereby raising `CONSTRAINT_ERROR`. This problem occurs when `Value` is less than any value in the array or greater than any value in the array. We can add a check for these conditions before beginning our search. But first we must make sure that the list is not empty. An attempt to compare `Value` with the first or last values in an empty list also raises `CONSTRAINT_ERROR`.

6. The assignment of values to the out parameters must be modified to satisfy the new postconditions.

```
-- Assign results to the out parameters
Success := Found;
if Found then
 Location := Midpoint;
else
 Location := Index_Type'Last;
end if;
```

Body 4-4 is the complete procedure body reflecting all these changes.

**Body 4-4—****Generic Binary Search Procedure Implements Specification 4-3**

```

procedure Binary_Search (List : in Array_Type;
 Value : in Element_Type;
 Success : out Boolean;
 Location : out Index_Type) is

 Found : Boolean; -- Has Value been found yet?
 Midpoint : Index_Type; -- Index of search area's midpoint
 First_Index : Index_Type; -- First index in current search area
 Last_Index : Index_Type; -- Last index in current search area

begin -- Binary_Search

 -- Is the list empty?
 if List'Length = 0 then
 Success := False;
 Location := Index_Type'Last;

 -- Is the value outside of the range of indices?
 elsif Value < List(List'First) or List(List'Last) < Value then
 Success := False;
 Location := Index_Type'Last;

 else -- Do the search.

 -- Initialize
 Found := False;
 First_Index := List'First;
 Last_Index := List'Last;

 -- Search until element found or the current search area is empty.
 -- Current search area is List (First_Index..Last_Index).
 while (First_Index <= Last_Index) and not Found loop
 -- Find middle element in the current search area.
 Midpoint := Index_Type'Val ((Index_Type'Pos (First_Index) +
 Index_Type'Pos (Last_Index)) / 2);
 -- Compare Value to middle element in search area.
 if List(Midpoint) = Value then
 Found := True;
 elsif Value < List(Midpoint) then
 Last_Index := Index_Type'Pred(Midpoint); -- Search 1st half
 else
 First_Index := Index_Type'Succ(Midpoint); -- Search 2nd half
 end if;
 end loop;
 end loop;
end procedure;

```

```

-- Assign results to the out parameters.
Success := Found;
if Found then
 Location := Midpoint;
else
 Location := Index_Type'Last;
end if;
end if;
end Binary_Search;

```

### Generalizing the Components

Because we declared the formal generic type `Element_Type` as range `<>`, our generic binary search procedure is limited to arrays whose components are whole numbers. The other choices we have discussed (`<<>` and `digits <>`) are also restrictive. We would like a generic binary search procedure that can be used with *any type* of component. We can do this by declaring the `Element_Type` as a generic formal private type. Here is a version of our generic procedure declaration that uses a private type definition:

```

generic
 -- Version 5

 type Element_Type is private;
 type Index_Type is (<>);
 type Array_Type is array (Index_Type range <>) of Element_Type;

 procedure Binary_Search (List : in Array_Type;
 Value : in Element_Type;
 Success : out Boolean;
 Location : out Index_Type);

```

In this declaration, the formal generic parameter `Element_Type` matches an actual parameter of *any* type. Thus we can instantiate procedures to search an array whose components are whole numbers, floating point numbers, characters, records, or any other type.

Because `Element_Type` may be any type, the operations that our binary search procedure can use with objects of this type are restricted to those available to *all* types. Only assignment (`:=`) and equality testing (`=` and `/=`) are available to all types. Looking at Body 4-4 for our procedure we see that it compares elements with both the `=` and the `<` operators. Here is that section of the procedure:

```

if List(Midpoint) = Value then
 Found := True;
elsif Value < List(Midpoint) then
 .
 .
 .

```

The Ada compiler reports a syntax error for using the `<` operator with a private type. The problem is that when we compile our generic binary search procedure we have no idea of what kind of elements could be in the array. The Ada compiler cannot supply the correct

< operator because future programmers can designate *any* type when they instantiate procedures from this template.

The solution, of course, is to have the programmer who instantiates a binary search procedure supply the comparison operation. This is done through a generic formal sub-program. In our case, we need a function that allows us to determine whether or not one element is less than another. Here is a version of our generic binary search procedure declaration that includes this generic formal subprogram:

```
generic -- Version 6

type Element_Type is private;
type Index_Type is (<>);
type Array_Type is array (Index_Type range <>) of Element_Type;
with function "<" (Left : in Element_Type;
 Right : in Element_Type) return Boolean;

procedure Binary_Search (List : in Array_Type;
 Value : in Element_Type;
 Success : out Boolean;
 Location : out Index_Type);
```

Let's look at some examples of how we might instantiate various search procedures from this generic binary search procedure.

```
type Month_Type is (January, February, March, April,
 May, June, July, August, September,
 October, November, December);

type Nat_Array is array (Month_Type range <>) of Natural;

procedure Nat_Search is new Binary_Search (Element_Type => Natural,
 Index_Type => Month_Type,
 Array_Type => Nat_Array,
 "<" => Standard."<");
```

Procedure `Nat_Search` searches an array of natural numbers. We have supplied the Ada's predefined < operator for type `Integer` (`Natural` is a subtype of `Integer`). It is not necessary to qualify this operator with package `Standard`; we have done so to make it clear that it is a predefined operator that we are supplying for the construction of a new procedure from the generic template.

Now let's look at how we might instantiate a binary search procedure to search an array of records.

```
subtype Name_String is String (1..20);
type GPA_Type is digits 4 range 0.00 .. 4.00;
type Student_Rec is
 record
 ID : Positive;
 Name : Name_String;
 GPA : GPA_Type;
 end record;
```

```

function "<" (Left : in Student_Rec;
 Right : in Student_Rec) return Boolean is
-- Compares two student record IDs
begin
 return Left.ID < Right.ID;
end "<";

type Stu_Array is array (Positive range <>) of Student_Rec;

procedure Stu_Search is new Binary_Search (Element_Type => Student_Rec,
 Index_Type => Positive,
 Array_Type => Stu_Array,
 "<" => "<");

```

Procedure `Stu_Search` searches an array of student records. Because there is no predefined `<` operator for record types, we have written our own. It compares the `ID` field of two student records. To meet the precondition of our binary search operation, the elements in our array of student records must be in ascending order by `ID`.

Although the Ada compiler finds no problems with this instantiation, running the tests described in Chapter 2 does. No search is successful! It would seem that the `=` operation in our binary search procedure always returns `False`. The `=` operation being used is that for records. Two student records are considered equal only if they have the same `ID`, the same `Name`, and the same `GPA`. For our search we consider the components equal when just the `ID` matches. We need to supply our own `"="` function. To do so, we declare our generic formal parameter `Element_Type` as `limited private` rather than `private`. This formal parameter still matches any actual parameter supplied. By limiting the type, we restrict the operations available to those supplied during instantiation as generic formal subprograms. We made this change so that we can define our own `"="` function.\* This addition is shown in Specification 4-3, our final version of the generic binary search procedure specification. Body 4-4 implements this specification.

We must now supply an actual function for the `"="` function. Here is the appropriate function for comparing two student records:

```

function Same_Student (Left : in Student_Rec;
 Right : in Student_Rec) return Boolean is
begin
 return Left.ID = Right.ID;
end Same_Student;

```

And here is the instantiation to create a procedure to search an array of student records:

```

procedure Stu_Search is new Binary_Search (Element_Type => Student_Rec,
 Index_Type => Positive,
 Array_Type => Stu_Array,
 "=" => Same_Student);
 "<" => "<");

```

---

\*In Ada 83 we can only define our own function `"="` for limited private types. This restriction is lifted in Ada 95; we can define our own function `"="` for any type.

### Specification 4-3— Generic Binary Search Procedure

```
generic

 type Element_Type is limited private;
 type Index_Type is (<>);
 type Array_Type is array (Index_Type range <>) of Element_Type;
 with function "=" (Left : in Element_Type;
 Right : in Element_Type) return Boolean;
 with function "<" (Left : in Element_Type;
 Right : in Element_Type) return Boolean;

 procedure Binary_Search (List : in Array_Type;
 Value : in Element_Type;
 Success : out Boolean;
 Location : out Index_Type);

-- Search List for Value

-- Preconditions: List is in ascending order

-- Postconditions: Value is in List and Success and
-- List(Location) = Value
-- or
-- Value is not in List and not Success and
-- Location = Index_Type'Last.
```

#### **Private Types and Generic Formal Private Types**

We have used the reserved word `private` in two different contexts: the declaration of private types and the declaration of generic formal types. Private types are used in an ADT package to prevent the program that uses the package from accessing the details of the type. These details are declared in the private part of the package specification and may be used in any way by the package body. By making the types private, we prevent a program that uses the package from accessing the details of the data structure. Then, if we change these details, we can be confident that we won't have to rewrite the application. You have just seen how generic formal private type parameters let us write a generic package that can be instantiated for any type. In this case, it is the package that is restricted in its use of the type. In order to handle any type, we can only use operations that are common to all types. The details of the type are declared outside of the generic package and may not be used in it.

It is easy to confuse these two different uses of *private*. Just remember that where the details of the type are declared is where they may be fully accessed. With a private type, the details of the type are defined *in the ADT package* and may be used without restriction by that package. With a generic formal private type parameter, the details of the actual type are defined *in the application* that instantiates a new package from the generic package. The details of the type are available to that program but not to the generic package.



## Summary

A set is an unordered collection of values. An item is either in the set (is a member) or not in the set (is not a member). Operations for the set come from mathematics. The most common operations are membership, union, intersection, difference, and subsets. In this chapter we developed a generic set package for sets whose elements are discrete types.

Packages allow us to encapsulate our implementation of a class. Generic packages are package templates that can be customized by the user of the class when they instantiate a package from our template. Ada also allows us to write generic subprograms. These make it easy to reuse a single procedure or function. We use generic parameters to supply the information necessary for the Ada compiler to carry out the customizing of generic packages and subprograms. The selection of a generic formal parameter type restricts (1) what type of actual parameter can be supplied by the user and (2) what operations the package may use with objects of that type.

Designing a generic package or subprogram for reuse takes more effort than designing it for a specific problem. We must select the most general generic parameters and write the body for the most general case. The benefits gained in reusing well-designed generic units outweigh the original design effort.

## Case Study:

### Minimizing Translations

Let's look at another problem whose solution involves sets. A European conference is being held in Brussels. The conference committee is designing signs to place outside of the different meeting rooms to help delegates find the correct rooms. There is not enough space on a sign to use every language spoken at the conference. Fortunately, the delegates are diplomats so most speak several different languages. The committee must find a small number of languages to put on the signs. These languages must be chosen so that every delegate can read at least one of them. Here is a more formal specification of this problem:

### Translations

**Function:** To find a small number of languages of which all delegates can read at least one.

**Input:** The names of the languages spoken by each delegate. Data is in a text file called `Delegate.Dat`. Each line of the file contains the data for one delegate.

**Output:** A listing of languages of which all delegates can read at least one.

We can model each delegate with a set of languages (a delegate either speaks or does not speak a particular language). To model all of the delegates, we can use an array of language sets. Here are the declarations needed:

```
Max_People : constant := 100; -- Maximum number of delegates

type Language_Type is (Dutch, English, Finnish, French, German, Italian,
 Norwegian, Portuguese, Spanish, Swedish);
-- Create a set class whose components are the languages.
package Language_Set is new Discrete_Set (Element_Type => Language_Type);
```

```

-- Types for an array of language sets
subtype Index_Type is Natural;
subtype Index_Range is Index_Type range 1..Max_People;
type Language_Set_Array is
 array (Index_Range range <>) of Language_Set.Set_Type;

-- A list of the languages spoken by each delegate. Each element
-- in the array is a set of languages spoken by one delegate.
Delegate_List : Language_Set_Array (Index_Range);
Delegate_Count : Index_Type;

```

If there is some common group of languages spoken by all the diplomats, we can find it by taking the intersection of all the language sets. For example, here are the language sets for three diplomats:

{Dutch French German} {English French German Spanish} {French Italian German}

The intersection of these three sets is

{French German}

This intersection is the set of languages spoken by all three diplomats. Either one of the languages in this intersection could be used alone for the signs. It is, however, likely that there is no language spoken by all the diplomats attending the conference. The following three language sets illustrate this possibility:

{Dutch French German} {English French Spanish} {Italian German}

The intersection of these sets is the empty set; there is no language common to these three diplomats.

Upon inspection of these last three sets, we can see that if each sign is written in both French and German, all the diplomats can read it. Such inspections are easy with only a few diplomats. It is much more difficult when there are hundreds of diplomats speaking dozens of languages.

### ***An Optimal Solution***

If the intersection of all of the diplomats' language sets yields an empty set, we can test different combinations of languages until we find a set that includes languages spoken by everyone. Start by testing all combinations of two different languages, then all combinations of three different languages, then all combinations of four different languages, and so on until we find a set of languages that satisfies all the diplomats. To test a potential set of languages, we determine the intersection of it with each diplomat's set of languages. If any intersection yields an empty set, that potential set of languages is inadequate. There are two problems with this solution. First, generating the various combinations is not a trivial algorithm. Second, even for a modest number of languages, the number of possible combinations can be very large. For example, the number of possible combinations of languages from a set of 25 is 33,554,431. If the conference has a hundred diplomats, our program may need to calculate over three billion set intersections.

### *A Greedy Solution*

A greedy algorithm is one that determines a satisfactory, but not necessarily best, solution to a problem in a short amount of time. For example, a greedy algorithm for our language problem might yield a set of five languages while the optimal algorithm yields a set of three languages. However, the number of calculations required in the greedy algorithm is much smaller. Here is a greedy algorithm for our language problem:

#### **Find Adequate Set of Languages**

```
Initialize Potential Set to the universal set of languages
I := first language in the universal set
loop
 Remove the Ith language from the Potential Set
 Test the Potential Set against all diplomats
 If the Potential Set does not satisfy some diplomat
 Put the Ith language back in the Potential Set
 end if
 exit loop when all languages have been tested.
 Increment I
end loop
```

In this algorithm we remove one language and test to see if the reduced set still satisfies all the diplomats. If the reduced set is inadequate for a single diplomat, we put that language back in. We do this for each language spoken. If we have 25 languages, the loop iterates 25 times. Testing 100 diplomats then requires the calculation of 2,500 intersections; far fewer than the potential three billion intersection calculations in the optimal algorithm. Program 4-1 implements this greedy algorithm.

#### **Program 4-1— Translations**

```
with Discrete_Set;
procedure Translations is

-- This program finds a "minimum" number of languages needed for an international
-- conference. Because a greedy algorithm is used, the languages selected
-- may not be the absolute minimum number.

Max_People : constant := 100; -- Maximum number of people

type Language_Type is (Dutch, English, Finnish, French, German, Italian,
 Norwegian, Portuguese, Spanish, Swedish);

-- Create a set class whose components are the languages.
package Language_Set is new Discrete_Set (Element_Type => Language_Type);
use Language_Set; -- Make set operations directly visible.
-- A package for I/O of languages
package Language_IO is new Text_IO Enumeration_IO (Enum => Language_Type);
```

```

-- Types for an array of language sets
subtype Index_Type is Natural;
subtype Index_Range is Index_Type range 1..Max_People;
type Language_Set_Array is array (Index_Range range <>) of Language_Set.Set_Type;

-- A list of the languages spoken by each delegate. Each element in the
-- array of this list is a set of languages spoken by one delegate.
Delegate_List : Language_Set_Array (Index_Range);
Delegate_Count : Index_Type;

Data_File : Text_IO.File_Type; -- File containing delegate language information
Min_Set : Language_Set.Set_Type; -- A small set of languages sufficient for all

procedure Put_Set (Set : in Language_Set.Set_Type) is
-- This procedure displays all of the languages in a given set.
begin
 for Language in Language_Type loop
 if Is_Member (Element => Language, Set => Set) then
 Language_IO.Put (Language);
 Text_IO.New_Line;
 end if;
 end loop;
end Put_Set;

function Adequate (Delegates : in Language_Set_Array;
 Set : in Language_Set.Set_Type) return Boolean is
-- This function determines whether or not the given Set of languages
-- is adequate for all the delegates. That is, is there at least one
-- language in the set for each delegate.
--
-- Preconditions: The array Delegates contains at least one element.

 Index : Index_Type;
 Inadequate : Boolean;

begin
 Index := Delegates'First;
 Inadequate := False;
 loop
 exit when Inadequate or Index > Delegates'Last;
 -- Set is inadequate if the delegate doesn't speak one of the languages.
 Inadequate := Delegates (Index) * Set = Language_Set.Empty_Set;
 Index := Index + 1;
 end loop;
 return not Inadequate;
end Adequate;

procedure Find_Minimum (List : in Language_Set_Array;
 Min_Set : out Language_Set.Set_Type) is
-- This procedure uses a greedy algorithm to determine some "small" set
-- of languages that satisfy the needs of all delegates in the List.

```

```

Languages : Language_Set.Set_Type; -- Local copy of Min_Set
Candidate : Language_Type; -- One language
begin -- Find_Minimum
 Languages := Language_Set.Universal_Set; -- Start with all languages
 -- Try to remove languages from the set
 Candidate := Language_Type'First;
 loop
 Languages := Languages - Candidate; -- Remove Candidate language from the set
 -- Is the current set of languages inadequate?
 if not Adequate (Delegates => List, Set => List, Set => Languages) then
 Languages := Languages + Candidate; -- Put the language back in
 end if;
 exit when Candidate = Language_Type'Last;
 Candidate := Language_Type'Succ (Candidate);
 end loop;
 Min_Set := Languages;
end Find_Minimum;

procedure Get_Delegate_Data (File : in out Text_IO.File_Type;
 Languages : out Language_Set.Set_Type) is
-- This procedure gets all the language data for one delegate.
--
-- Preconditions: The languages spoken by this delegate are on a single line.
-- The line terminator follows immediately after the last language.

 Delegate : Language_Set.Set_Type; -- Local copy of out parameter Languages
 Language : Language_Type; -- One language spoken by a delegate
begin
 Delegate := Language_Set.Empty_Set; -- Initialize delegate's set of languages
 Language_Loop: -- Each iteration, one language for a delegate is obtained
 loop
 exit Language_Loop when Text_IO.End_Of_Line (File);
 -- Get a language and add it to the delegate's language set
 Language_IO.Get (File => File, Item => Language);
 Delegate := Delegate + Language;
 end loop Language_Loop;
 Languages := Delegate;
end Get_Delegate_Data;

begin -- Program Translations

 -- Get the language data for all the Delegate_List

 Text_IO.Open (File => Data_File, Name => "Delegate.Dat", Mode => Text_IO.In_File);
 Delegate_Count := 0;
 Delegate_Loop: -- Each iteration, the language data for one delegate is obtained.
 loop
 exit Delegate_Loop when Text_IO.End_Of_File (Data_File);
 Delegate_Count := Delegate_Count + 1;
 Get_Delegate_Data (File => Data_File, Languages => Delegate_List (Delegate_Count));
 Text_IO.Skip_Line (Data_File);
 end loop Delegate_Loop;

```

```

Text_IO.Close (Data_File);
-- See if there are any languages spoken by ALL delegates by
-- determining the intersection of all their language sets.
Min_Set := Delegate_List (1); -- Initialize to 1st delegate's languages
for Count in 2..Delegate_Count loop
 Min_Set := Min_Set * Delegate_List (Count);
end loop;

if Min_Set /= Language_Set.Empty_Set then -- Is there one or more common languages?
 Text_IO.Put_Line ("Use any one of the following languages.");
else
 Text_IO.Put_Line ("Use all of the following languages.");
 -- Find some small set that satisfies all delegates
 Find_Minimum (List => Delegate_List (1..Delegate_Count),
 Min_Set => Min_Set);
end if;

Text_IO.New_Line;
Put_Set (Min_Set);

end Translations;

```

### Exercises

1. Write a body for `Bingo_Basket` that uses a set of numbers that have been drawn rather than a set of numbers that have not been drawn.
2. (a) Using the generic discrete set ADT package developed in this chapter, write the necessary declarations to instantiate a set class to contain the months of the year.
  - (b) Using the package you instantiated in part (a), declare variables `Winter_Months`, `Fall_Months`, and `Cool_Months` to be set objects (variables).
  - (c) Write a code fragment to assign the appropriate months to `Winter_Months`.
  - (d) Write a code fragment to assign the appropriate months to `Fall_Months`.
  - (e) Write a single statement to assign the union of `Winter_Months` and `Fall_Months` to `Cool_Months`.

3. (a) Add the following array type and operation to the specification of package `Discrete_Set` and implement the body of the function

```

-- An array of set elements may be converted to type Set_Type.
type Set_Array is array (Positive range <>) of Element_Type;

function To_Set (Array_Of_Elements : in Set_Array) return Set_Type;
-- This function returns a set consisting of the elements
-- in the given array.

```

- (b) Use function `To_Set` to write a single Ada statement to assign the appropriate months to the set object `Winter_Months` you declared in the previous question. Hint: use an array aggregate.
- (c) Should type `Set_Array` be declared as a private type like `Set_Type`? Explain your answer.

4. Add an operation to the specification of package `Discrete_Set` to determine the cardinality of a set (the number of elements in a given set). Implement the body of this operation.

5. Add the following operation to the specification of package `Discrete_Set`.

```
procedure Put_Set (Set : in Set_Type);
-- Display all the elements in Set, one per line.
```

Implement the body of this procedure. Hint: use the `'Image` attribute to convert a discrete value to a string and then use `Text_IO.Put_Line` to display it.

6. A programmer used the following instantiation in a program:

```
package Integer_Set is new Discrete_Set (Element_Type => Integer);
```

The program compiles without error, but when it is run, `STORAGE_ERROR` is raised. Upon single stepping through the program with the debugger, the programmer discovered that this exception is raised by the following variable declaration:

```
Nums : Integer_Set.Set_Type; -- A set of Integers
```

Explain why `STORAGE_ERROR` is raised here.

7. Why did we choose to implement the set as an abstract data type (ADT) rather than as an abstract data object (ADO)?

8. The `Draw` operation of the `Bingo_Basket` ADT we developed does not use a very efficient algorithm. Numbers between 1 and 75 are repeatedly chosen at random until a check reveals that the chosen number is still "in the basket." As the number of numbers remaining in the basket becomes small, this loop may execute many times before a chosen number is found in the basket. Develop a more efficient algorithm. Hint: use an array of Bingo numbers instead of a set of Bingo numbers and use values from the random number generator to "shuffle" the values in the array.

9. Explain the difference between a private type and a generic formal private type.

10. (a) Write a specification for a generic procedure called `Swap` that swaps the contents of two discrete variables.

(b) Write some declarations to illustrate an instantiation of the generic procedure whose specification you wrote in part (a).

(c) Write the body for generic procedure `Swap` whose specification you wrote in part (a).

11. (a) Write a specification for a generic function called `Max` that returns a value equal to the larger of two discrete parameters.

(b) Write some declarations to illustrate an instantiation of the generic function whose specification you wrote in part (a).

(c) Write the body for generic function `Max` whose specification you wrote in part (a).

(d) Rewrite the specification from part (a) to make the function as general as possible. (Design it for reuse.)

(e) Write some declarations to illustrate an instantiation of the generic procedure whose specification you wrote in part (d).

12. (a) Write a specification for a generic procedure called `Maximum` that accepts an unconstrained array (indexed by type `Integer`) of discrete values and returns the largest.

(b) Write some declarations to illustrate an instantiation of the generic procedure whose specification you wrote in part (a).

(c) Write the body for generic procedure `Maximum` whose specification you wrote in part (a).

(d) Rewrite the specification from part (a) to make the procedure as general as possible. (Design it for reuse.)

- (e) Write some declarations to illustrate an instantiation of the generic procedure whose specification you wrote in part (d).
- (f) What changes, if any, must be made to the body you wrote in part (c) for the more general specification you wrote in part (d)?

### Programming Problems

1. Sorting is a very common operation.
  - (a) Using a sort algorithm of your choice, write a generic sort procedure specification. Design it for reuse.
  - (b) Write the body for your generic sort procedure.
  - (c) Write a test plan and driver program to test your generic sort procedure.
2. All possible languages spoken at the conference described in the case study at the end of this chapter were coded directly into the Ada program. Modify this program so that the program reads the names of all the languages spoken as strings before the languages spoken by each delegate are read. One way to accomplish this modification is to associate each language with an integer (the first language entered is 1, the second 2 and so on). After the languages have been entered you can instantiate a set package for sets of the size needed. Why can't you just instantiate a set of `Positive` numbers? Design a package to handle the association between languages and numbers. Think carefully about what operations to include and whether it should be an ADT or an ADO.



## 5— Strings

### GOALS

After reading this chapter, you should be able to

- use a string class in an application
- differentiate between fixed-length and varying-length string classes
- differentiate between bounded-length and unbounded-length string classes
- implement the different string classes
- declare and use access variables to designate dynamically created objects
- use Ada's allocator `new` to create space for a new object at run time
- understand the difference between an access object and a designated object
- use procedures instantiated from `Unchecked_Deallocation` to reclaim memory used by a dynamic object

## String Terminology

Computers are used extensively to manipulate text. You use an editor program to enter and modify your Ada programs and you probably use a word processing program to enter, modify, and print essays for your English class. Each of these applications has its own set of operations for manipulating the text. For example, while a word processing program wraps the text around to the next line when you reach the end of a line, a program editor does not.

We can find many different classes in a typical essay: sections, paragraphs, lines, sentences, words, and characters. What abstractions can we use to model all of the requirements in these different applications? From your previous study of Ada, you are familiar with two useful predefined classes: the character and the string. The declaration of `String` given in package `Standard` is

```
type String is array (Positive range <>) of Character;
```

---

### Fixed-Length String

A string that contains a fixed (constant) number of characters.

---

From this declaration we can see that type `String` is an unconstrained array of characters. Typically, we declare constrained array subtypes from type `String` that we then use to declare string objects. Because the number of characters in an array *object* is fixed, string objects based on type `String` are known as **fixed-length strings**. As with any array type, the primitive operations of this string type are meager; we can index, slice, and concatenate. Ada 95 provides over 20 additional operations for type `String` in the package `Ada.Strings.Fixed`.

---

### Varying-Length String

A string in which the number of characters can change.

### Bounded-Length String

A varying-length string with a fixed *maximum* number of characters. A bounded-length string has a current length and a maximum length.

### Unbounded-Length String

A varying-length string with no bound on the number of characters.

---

In most problems, the number of characters we need to keep together as a unit is not constant. For example, the number of characters on a line in an Ada program can vary from zero to 80 (the typical width of a display screen) or more. A string whose length can change is called a **varying-length string**. Ada 95 has two predefined varying-length string classes. Package `Ada.Strings.Bounded` defines a **bounded-length string** type. A bounded-length string object has both a current length, which can change, and a maximum length, which cannot change. Package `Ada.Strings.Unbounded` defines an **unbounded-length string** type. Like the bounded-length string, the length of an unbounded string object can change over the object's lifetime. Unlike the bounded-length string, there is no maximum size associated with an unbounded-length string object.

Why would we choose to use a bounded-length string when an unbounded-length string type is available? Generally, unbounded-length strings cannot be implemented to run as efficiently as bounded-length strings; the operations take more time to execute. Later in this chapter we discuss other problems that a programmer must tackle when using unbounded-length strings.

## The Abstract Level

What are the basic text string operations? Operations for the set class developed in Chapter 4 come from mathematics. There is widespread agreement on what constitutes the basic operations of this class. There is no such agreement with text strings. Ada 95 defines nearly 50 operations for bounded-length strings in the package `Ada.Strings.Bounded` and over 20 operations for fixed-length strings in the package `Ada.Strings.Fixed`. Ada 83 supplies only the basic array operations (indexing, slicing, and concatenation) for fixed-length

strings. It is however, not difficult to construct our own bounded-length string class for either version of Ada.

### *A Bounded-Length String Specification*

Specification 5-1 describes a bounded-length string ADT. There are five groups of operations in this specification. The first group contains relational operators for comparing bounded strings. We have overloaded the traditional operators for these operations. The second group contains operations for combining strings and characters. We have included catenation functions for several combinations of characters, fixed-length strings, and bounded-length strings. The third group includes functions for determining how many characters are in a string, for extracting portions of a string, and searching for some pattern in a string. The fourth group contains functions for converting between Ada's predefined fixed-length string type `String` and our bounded-length string type. The last group contains procedures for the input and output of bounded-length strings.

#### **Specification 5-1— Bounded-Length Strings**

```
with Text_IO;
generic

 Max_Length : Positive; -- The maximum size of an individual Bounded_String

package Bounded_Strings is

 -- This package implements a bounded-length string type.

 type Bounded_String is private; -- Bounded-length string type

 Null_Bounded_String : constant Bounded_String; -- A string with no characters

 -- Relational operators

 function "<" (Left : in Bounded_String; Right : in Bounded_String) return Boolean;
 function "<=" (Left : in Bounded_String; Right : in Bounded_String) return Boolean;
 function ">" (Left : in Bounded_String; Right : in Bounded_String) return Boolean;
 function ">=" (Left : in Bounded_String; Right : in Bounded_String) return Boolean;
 function Equal (Left : in Bounded_String; Right : in Bounded_String) return Boolean;

 -- Catenation operators

 -- Exceptions: CONSTRAINT_ERROR is raised if the combined length of the two
 -- parameters exceeds the value of Max_Length

 function "&" (Left : in Bounded_String; Right : in Bounded_String)
 return Bounded_String;
 function "&" (Left : in Bounded_String; Right : in Character)
 return Bounded_String;
```

```

function "&" (Left : in Character; Right : in Bounded_String)
 return Bounded_String;
function "&" (Left : in Bounded_String; Right : in String)
 return Bounded_String;
function "&" (Left : in String; Right : in Bounded_String)
 return Bounded_String;

function Length (Source : in Bounded_String) return Natural;
-- This function returns the number of characters in a string.
--
-- Preconditions: None
--
-- Postconditions: The number of characters in Source is returned

function Slice (Source : in Bounded_String;
 Low : in Positive;
 High : in Natural) return Bounded_String;
-- This function returns a substring of the given string
--
-- Preconditions: None
--
-- Postconditions: If Low <= High, a string composed of the characters from
-- position Low to position High is returned
-- Otherwise, a null string is returned
--
-- Exceptions: CONSTRAINT_ERROR is raised if Low <= High and
-- High is greater than Length (Source)

function Element (Source : in Bounded_String;
 Index : in Positive) return Character;
-- This function returns the character in the string at the position given by Index.
--
-- Preconditions: None
--
-- Postconditions: The character in Source at Position is returned
--
-- Exceptions: CONSTRAINT_ERROR is raised if Index is
-- greater than Length (Source)

function Index (Source : in Bounded_String;
 Pattern : in Bounded_String) return Natural;
-- This procedure searches for Pattern in Source.
--
-- Preconditions: None
--
-- Postconditions: If the Pattern is found within Source, Index returns the smallest
-- index I such that the slice of Source starting at I matches
-- Pattern
-- Otherwise, 0 is returned

```

```

-- Conversions

function To_String (Source : in Bounded_String) return String;
-- This function converts a bounded-length string to a fixed-length string.
--
-- Preconditions: None
--
-- Postconditions: The characters in Source are returned as
-- an array of characters
--

function To_Bounded_String (Source : in String) return Bounded_String;
-- This function converts a fixed-length string to a bounded-length string.
--
-- Preconditions: None
--
-- Postconditions: The characters in Source are returned
-- as a bounded-length string
--
-- Exceptions: CONSTRAINT_ERROR is raised if
-- Source'Last > Max_Length
--

-- Input and Output Operations

procedure Get_Line (File : in Text_IO.File_Type := Text_IO.Current_Input;
 Item : out Bounded_String);
-- Bounded-length string equivalent to Text_IO.Get_Line

procedure Put (File : in Text_IO.File_Type := Text_IO.Current_Output;
 Item : in Bounded_String);
-- Bounded-length string equivalent to Text_IO.Put

private

-- We fill this part in later.

end Bounded_Strings;

```

Our bounded-length string package is a generic package. The single generic formal parameter, `Max_Length`, is different than the one we used in our generic set package. This parameter is a *generic formal object*. Generic formal objects are much like subprogram `in` mode parameters. A value is given when the package is instantiated. Within the generic package, that value may be used as a constant is used. Here is how we would instantiate a package for bounded-length strings with maximum sizes of 132 and 40 characters:

```

package String132 is new Bounded_Strings (Max_Length => 132);
package String40 is new Bounded_Strings (Max_Length => 40);

```

And here are examples of declarations of bounded-length string variables that can contain a maximum of 132 and 40 characters:

```
Output_Line : String132.Bounded_String;
Last_Name : String40.Bounded_String;
```

Notice that these two variables are not the same type. We must use operations from package `String132` to process `Output_Line` and operations from package `String40` to process `Last_Name`. We can, however, convert from one type to another through an intermediate type—the standard fixed-length type `String`. Here is a function using this technique that converts a `String40` bounded-length string to a `String132` bounded-length string:

```
function To_String132 (Value : in String40.Bounded_String)
 return String132.Bounded_String is
begin
 return String132.To_Bounded_String (String40.To_String (Value));
end To_String132;
```

The call to function `String40.To_String` converts `Value` to a fixed-length string, and the call to `String132.To_Bounded_String` converts that fixed-length string to a (`String132`) bounded-length string. The qualification of these function calls can be eliminated through the use of a `use` clause. The algorithm that converts a `String132` string to a `String40` string must somehow handle the potential problem of converting a `String132` string that is too long for a `String40` string. We can choose to shorten the string by truncating the "extra" characters or simply allow the run-time system to raise `CONSTRAINT_ERROR`. Exercise 3 at the end of this chapter asks you to write such a function.

### The Application Level

As we noted earlier, there are many applications for strings. We use a simple problem to illustrate the use of our bounded-length string class.

#### *Alphabetizing Names*

A user enters names (one per line) at the keyboard. After all the names are entered, our program displays them in alphabetical order. Names may be entered in one of two forms: a first name followed by a last name or a last name followed by a first name. In the first case, first and last names are separated by a single space. When a last name is entered first, it is separated from the first name by a comma and a single space. Here is an example of each of these forms:

Mildred Smedley

Beasley, Horace

The program should alphabetize and display all names in the second form (last name first). The user indicates that all names have been entered by entering a null name, a line with no characters. To keep our example short, we make the following assumptions:

1. There are no more than 20 names.
2. All input lines contain less than 30 characters.
3. There is exactly one blank between the first and last names.
4. There are no leading or trailing blanks.
5. There are no blanks within a first name or a last name.

Here is the algorithm we developed to solve this problem:

### Sort Names

```

Num Names := 0
loop
 Get Name
 exit when the length of Name is 0
 Convert Name to last name first form
 Increment Num Names
 Name List (Num Names) := Name
end loop
Sort Name List
Display Name List

```

### Convert to Last Name First (Name : in out)

```

Find the blank in Name
if the character before the blank is not a comma
 Name := Portion of name after blank & "," & Portion of Name before blank
end if

```

We can implement the names in this algorithm as bounded-length strings (using our bounded-length string class) or as fixed-length strings (using Ada's predefined type `String`). If we use fixed-length strings we need to pad every name with blanks and write our code so these blanks are ignored. Program 5-1 implements this algorithm using our bounded-length string class.

### Program 5-1—

#### Sort Names (Bounded-Length Strings)

```

with Bounded_Strings;
with Text_IO;
procedure Sort_Names is

-- This program reads names from the keyboard and displays them in alphabetical
-- order. Names may be entered one per line in one of two forms:
--
-- First_Name Last_Name or Last_Name, First_Name
--

```

```

-- The comma is required in the second form.
-- End of input is indicated with a null name.
--
-- Assumptions: 1. There are no more than 20 names.
-- 2. All input lines contain less than 30 characters.
-- 3. There is exactly one blank between the first and last names.
-- 4. There are no leading or trailing blanks.
-- 5. There are no blanks within a first name or a last name.

Max_Name_Length : constant := 30; -- Maximum characters in any name
Max_Names : constant := 20; -- Maximum number of names

-- Instantiate a bounded-length string package
package String30 is new Bounded_Strings (Max_Length => Max_Name_Length);
use String30; -- Make operations in String30 directly visible

-- Array type for holding all the names
type Index_Type is range 0..Max_Names;
subtype Index_Range is Index_Type range 1..Max_Names;
type Name_Array is array (Index_Range range <>) of String30.Bounded_String;

-- Variables for main program
Name : String30.Bounded_String; -- One name
Name_List : Name_Array (Index_Range); -- List of names entered by the user
Num_Names : Index_Type; -- Number of names entered

procedure Selection_Sort (List : in out Name_Array) is

-- An unordered list is taken as input. The same list
-- with the components in ascending order is returned.

 Name : String30.Bounded_String; -- Temporary variable for swapping
 Min_Index : Index_Type; -- Index of minimum so far

begin -- Selection_Sort
 for Pass_Count in List'First .. List'Last - 1 loop
 -- Find the index of the smallest item in List(Pass_Count)..List(List'Last)
 Min_Index := Pass_Count;
 for Place_Count in Pass_Count + 1 .. List'Last loop
 -- Compare the two names with String30."<"
 if List(Place_Count) < List(Min_Index) then
 Min_Index := Place_Count;
 end if;
 end loop;
 -- Swap List(Min_Index) and List(Pass_Count)
 Name := List(Min_Index);
 List(Min_Index) := List(Pass_Count);
 List(Pass_Count) := Name;
 end loop;
end Selection_Sort;

```



```

procedure Last_Name_First (Name : in out String30.Bounded_String) is
-- This procedure converts a name from the form
-- First_Name Last_Name to Last_Name, First_Name.
-- If the name is already last name first, it is not changed.
--
-- Preconditions: There is exactly one blank between the first and last
-- names. If last name is first, it is followed by a
-- comma.

 Blank : constant Bounded_String := To_Bounded_String (" ");

 Position : Natural; -- Position result of string search

begin
 -- Determine the position of the blank
 Position := Index (Source => Name, Pattern => Blank);
 -- Is the first name first (no comma indicates it is)?
 if Element (Name, Position-1) /= ',' then
 -- Put the last name first (separated by a comma and a blank)
 Name := Slice (Name, Position+1, Length(Name)) -- The last name
 & ", " -- A comma and blank
 & Slice (Name, 1, Position-1); -- The first name
 end if;
end Last_Name_First;

begin -- Sort_Names
 -- Get all the names
 Num_Names := 0;
 Input_Loop: -- Each iteration, one name is processed
 loop
 Text_IO.Put_Line ("Enter a name (blank name to end)");
 String30.Get_Line (Item => Name);
 -- Is this a null name?
 exit Input_Loop when Length (Name) = 0;
 -- Make sure the last name comes before the first name
 Last_Name_First (Name);
 -- Put the name into the list
 Num_Names := Num_Names + 1;
 Name_List (Num_Names) := Name;
 end loop Input_Loop;

 -- Sort the names
 Selection_Sort (List => Name_List (1..Num_Names));

 -- Display the sorted names
 for Index in 1..Num_Names loop
 String30.Put (Item => Name_List (Index));
 Text_IO.New_Line;
 end loop;

end Sort_Names;

```

Notice our use in this program of the `Element`, `Get_Line`, `Index`, `Length`, `Put`, `Slice`, `"&"`, and `"<"` operations from our bounded-length string class. These are highlighted in the program.

### Creating Additional Operations

Once you begin designing programs to manipulate strings, you will soon discover that our bounded-length string class contains only the most basic operations. Ada 95's bounded-length string class (package `Ada.Strings.Bounded`) contains nearly 50 string operations. We could add additional operations to our class. But, in order to give you some experience in using the existing operations, we choose instead to create the additional operators using the operations already available.

The first operation we develop removes a portion of a string. Here is a declaration for this procedure:

```
procedure Delete (Str : in out Bounded_String;
 Low : in Positive;
 High : in Positive);
-- Removes the characters from position Low to position High

-- Preconditions: None

-- Postconditions: Str is the original Str with the characters from
-- position Low to position High removed. No
-- characters are removed if High < Low.

-- Exceptions: CONSTRAINT_ERROR if Low <= High and
-- High > Length (Str).
-- Str is unchanged.
```

The algorithm we use to delete the substring does so by extracting the substrings on either side of the substring we wish to remove and combining them to form the result string. Here is the body of the procedure:

```
procedure Delete (Str : in out Bounded_String;
 Low : in Positive;
 High : in Positive) is
begin
 Str := Slice (Source => Str, Low => 1, High => Low-1) &
 Slice (Source => Str, Low => High+1, High => Length(Str));
end Delete;
```

The name-sorting application we developed in the last section was very explicit about the form of the input strings. There could be no leading or trailing blanks. And there had to be exactly one blank between the first and last names. Let's develop an operation to help ease the second restriction. This operation replaces all occurrences of multiple blanks with a single blank. Here is its declaration:

```
procedure Remove_Extra (Str : in out Bounded_String);
-- Removes extra blanks

-- Preconditions: None
```

```
-- Postconditions: All multiple blanks in Str are replaced
-- with a single blank
```

One approach to implementing this operation is to use `Bounded_Strings.Element` to examine individual characters. We could start with the first character, the second character, and so on until we found a blank. Then we would need to find out if there were any additional blanks just after it. This sort of search sounds similar to the description of `Bounded_Strings.Index`. Isn't there some way to make use of the procedure we already have? We can search for a pattern consisting of two blanks. If we find one, we delete one of those blanks. We continue searching and deleting until our search is unsuccessful. At this point there are no multiple blanks remaining in the string. If we enter four blanks between the first name and the last name, this algorithm would make three iterations to delete three of them. How do we delete a blank? We can use the `Delete` procedure we just wrote. Here is the body of `Remove_Extra`:

```
procedure Remove_Extra (Str : in out Bounded_String) is
 Blank_Pos : Natural;
 Double_Blank : constant Bounded_String := To_Bounded_String (" ");
begin
 -- Each iteration one blank is removed
 loop
 Blank_Pos := Index (Source => Str, -- Look for a double blank
 Pattern => Double_Blank);
 exit when Blank_Pos = 0; -- Exit when no double blank found
 -- Delete the first blank of the pair
 Delete (Str => Str, Low => Blank_Pos, High => Blank_Pos);
 end loop;
end Remove_Extra;
```

Although the implementation we might develop for these operations *within* the `Bounded_Strings` package might be more efficient than these, we have shown that we can create our own operations from those already present.

### The Implementation Level

We did not include the private part declarations for our bounded string type in Specification 5-1. We present them here:

```
private

subtype Length_Range is Natural range 0..Max_Length;
subtype Index_Range is Length_Range range 1..Max_Length;

subtype String_Array is String (Index_Range);

type Bounded_String is
 record
 Length : Length_Range := 0; -- The number of characters
 Data : String_Array; -- The characters in the string
 end record;
```

```
Null_Bounded_String : constant Bounded_String := (0, (others => ' '));
```

```
end Bounded_Strings;
```

The record for the bounded-length string type contains a field with the number of characters currently in the string and a field that is a fixed-length string that holds the characters. Only the characters from position 1 to `Length` in this array contain valid data. The remaining characters in the fixed-length string field are undefined. Figure 5-1 illustrates this record type. Body 5-1 is the body of package `Bounded_Strings`, in which most of the operations are implemented as simple Ada statements involving array operations (both constrained and unconstrained), slices, and `Text_IO`.

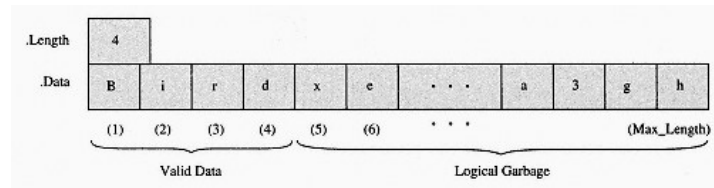


Figure 5-1  
A Bounded-Length String

### Body 5-1— Bounded-Length Strings Implements Specification 5-1

```
package body Bounded_Strings is
```

```
-- This package implements a bounded-length string type.
```

```

-- Relational operators

```

```
function "<" (Left : in Bounded_String; Right : in Bounded_String) return Boolean is
begin
 return Left.Data(1..Left.Length) < Right.Data(1..Right.Length);
end "<";
```

```
function "<=" (Left : in Bounded_String; Right : in Bounded_String) return Boolean is
begin
 return Left.Data(1..Left.Length) <= Right.Data(1..Right.Length);
end "<=";
```

```
function ">" (Left : in Bounded_String; Right : in Bounded_String) return Boolean is
begin
 return Left.Data(1..Left.Length) > Right.Data(1..Right.Length);
end ">";
```

```

function ">=" (Left : in Bounded_String; Right : in Bounded_String) return Boolean is
begin
 return Left.Data(1..Left.Length) >= Right.Data(1..Right.Length);
end ">=";

function Equal (Left : in Bounded_String; Right : in Bounded_String) return Boolean is
begin
 return Left.Data(1..Left.Length) = Right.Data(1..Right.Length);
end Equal;

-- Catenation operators

function "&" (Left : in Bounded_String; Right : in Bounded_String)
 return Bounded_String is
 Result : Bounded_String;
begin
 Result.Length := Left.Length + Right.Length;
 Result.Data(1..Left.Length) := Left.Data(1..Left.Length);
 Result.Data(Left.Length+1..Result.Length) := Right.Data(1..Right.Length);
 return Result;
end "&";

function "&" (Left : in Bounded_String; Right : in Character) return Bounded_String is
 Result : Bounded_String;
begin
 Result.Length := Left.Length + 1;
 Result.Data(1..Left.Length) := Left.Data(1..Left.Length);
 Result.Data(Result.Length) := Right;
 return Result;
end "&";

function "&" (Left : in Character; Right : in Bounded_String) return Bounded_String is
 Result : Bounded_String;
begin
 Result.Length := Right.Length + 1;
 Result.Data(1) := Left;
 Result.Data(2..Result.Length) := Right.Data(1..Right.Length);
 return Result;
end "&";

function "&" (Left : in Bounded_String; Right : in String) return Bounded_String is
 Result : Bounded_String;
begin
 Result.Length := Left.Length + Right.Length;
 Result.Data(1..Left.Length) := Left.Data(1..Left.Length);
 Result.Data(Left.Length+1..Result.Length) := Right;
 return Result;
end "&";

function "&" (Left : in String; Right : in Bounded_String) return Bounded_String is
 Result : Bounded_String;

```

```

begin
 Result.Length := Left.Length + Right.Length;
 Result.Data(1..Left.Length) := Left;
 Result.Data(Left.Length+1..Result.Length) := Right.Data(1..Right.Length);
 return Result;
end "&";

```

```

function Slice (Source : in Bounded_String;
 Low : in Positive;
 High : in Natural) return Bounded_String is
 Result : Bounded_String;
begin
 if High < Low then
 -- Null string?
 Result.Length := 0;
 elsif High <= Source.Length then
 -- Normal case?
 Result.Length := High - Low + 1;
 Result.Data(1..Result.Length) := Source.Data(Low..High);
 else
 -- Range out of bounds
 raise CONSTRAINT_ERROR;
 end if;
 return Result;
end Slice;

```

```

function Length (Source : in Bounded_String) return Natural is
begin
 return Source.Length;
end Length;

```

```

function Element (Source : in Bounded_String;
 Index : in Positive) return Character is
begin
 return Source.Data(Index);
end Element;

```

```

function Index (Source : in Bounded_String;
 Pattern : in Bounded_String) return Natural is

 First : Positive;
 Last : Positive;

 -- Pattern is compared to
 -- Source (First..Last)

begin
 -- Search
 First := 1;
 Last := Pattern.Length;
 Search_Loop: -- Each iteration, the pattern is compared to one slice
 loop
 -- of the source
 -- Exit when there are fewer characters remaining in Source to check
 -- than there are in the Pattern or when match is found
 exit when Last > Source.Length or else
 Source.Data(First..Last) = Pattern.Data(1..Pattern.Length);
 end loop;

```

```

 First := First + 1;
 Last := Last + 1;
 end loop Search_Loop;

 if Last <= Source.Length then -- Did we find the pattern?
 return First;
 else
 return 0;
 end if;
end Index;

function To_String (Source : in Bounded_String) return String is
begin
 return Source.Data(1..Source.Length); -- return the slice
end To_String;

function To_Bounded_String (Source : in String) return Bounded_String is
 Result : Bounded_String;
begin
 Result.Length := Source'Last - Source'First + 1;
 Result.Data(1..Result.Length) := Source;
 return Result;
end To_Bounded_String;

-- Input and Output Operations

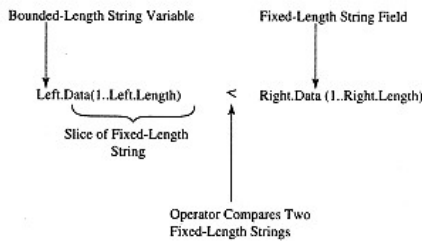
procedure Get_Line (File : in Text_IO.File_Type := Text_IO.Current_Input;
 Item : out Bounded_String) is
begin
 Text_IO.Get_Line (File => File, Item => Item.Data, Last => Item.Length);
end Get_Line;

procedure Put (File : in Text_IO.File_Type := Text_IO.Current_Output;
 Item : in Bounded_String) is
begin
 Text_IO.Put (File => File, Item => Item.Data (1..Item.Length));
end Put;

end Bounded_Strings;

```

To compare two bounded-length strings we compare the *valid* characters in the two `Data` fields. These fields are type `String`, Ada's predefined fixed-length string type. Ada defines slicing and relational operators for this type that we can use in our comparison functions. Here, for example, is a Boolean expression that evaluates to `True` if the bounded-length string `Left` is less than `Right`:



Each of the catenation operators declares a local variable for its result. The `Length` field of the result is calculated as the sum of the lengths of the two parameters. The length of a bounded-length string parameter is obtained from its `Length` field. The length of a character parameter is always 1. The length of a fixed-length String parameter is obtained through a `'Length` array attribute. Recall that `'Length` returns the number of elements in an array. We use slices of the `Data` fields and the predefined fixed-length String catenation operators to assign values to the `Data` fields of the results.

Where is `CONSTRAINT_ERROR` raised when the sum of the lengths of the two parameters of a catenation operator exceed `Max_Length`? There is no need for us to raise this exception explicitly. We have taken advantage of Ada's array bound-checking at run time. When the function attempts to assign more characters to `Result.Data` than it can hold (`Max_Length`), the Ada run-time system raises `CONSTRAINT_ERROR`.

The `Index` operation requires a little more effort to understand. To locate a pattern within a string by hand, you would try to match the pattern starting with the first character of the string. If no match is found, you would attempt to match it starting at the second character of the string. If still no match is found, you would attempt to match it starting at the third character of the string. You would continue this process until you either find the pattern or it is longer than the characters remaining in the string. Figures 5-2 and 5-3 illustrate this algorithm.

### *Analysis of the String Algorithms*

There is only one loop in the bounded-length string package. This loop is the search loop in `Index` that we just finished examining. Let's define our problem size as the number of characters in a string. Looking at this loop in `Index`, you can see that the longer the string is the more iterations we have to make to reach the end of it. This loop is  $O(N)$ . Within this loop we compare two array slices—an array operation whose execution time is proportional to the size of the slice. The size of the slice in this comparison is the number of characters in `Pattern`. If we use  $M$  to represent the size of the pattern, we can classify `Index` as  $O(N*M)$ . In most searches, the pattern is usually much smaller than the string being searched so that we can ignore  $M$ . We classify `Index` as  $O(N)$ .

The slice operation, conversion operations, and all of the relational and catenation operators are implemented with predefined array slice operations. The amount of time to execute any of these operations is proportional to the size of the strings. All of these operators are therefore  $O(N)$ .

`Length` and `Element` do not involve loops or array operations. They are  $O(1)$ . It always takes the same amount of time to determine the length or retrieve one character no matter how long the string is.



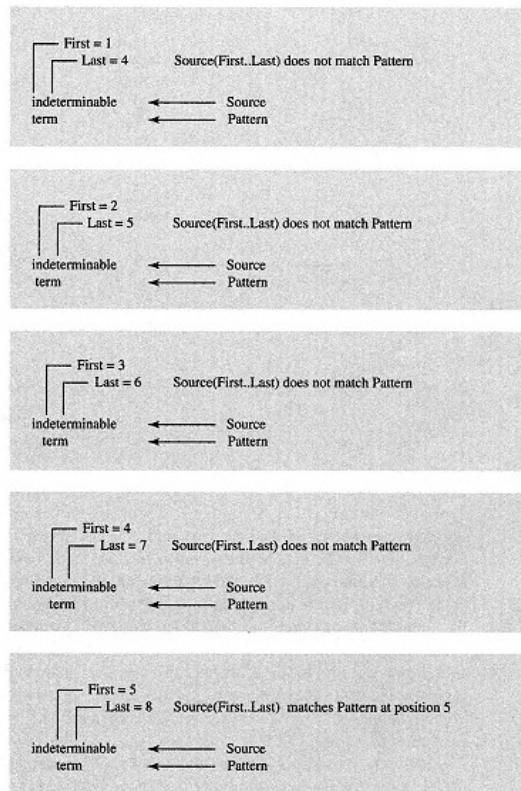


Figure 5-2  
Example of Successful String Search

### Ada 83 Implementation Note

We have chosen to make `Bounded_String` a private type rather than a limited private type to permit use of the assignment operator (`:`) with bounded-length string objects. This choice presents a problem. In Ada 83 we only can define our own `=` operator for limited private types. Because we have chosen to make `Bounded_String` a private type, we had to name our Boolean function for equality something else (we used the name `Equal`). The user of this package must use this `Equal` function instead of the primitive `=` operator that Ada defines for the private type. Using `=` returns incorrect results. Here's why. Our private type is a record. For two records to be considered equal, every field in one record must be equal to the corresponding field in the other record. One of the fields in

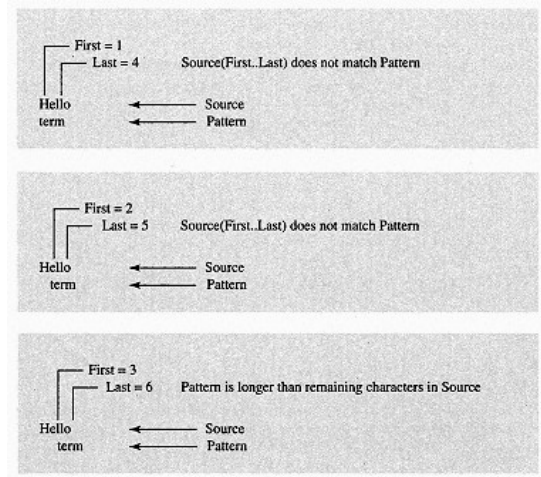


Figure 5-3  
Example of Unsuccessful String Search

`Bounded_String` is an array (`String` is an array of characters). All of the characters in one array must be equal to the corresponding characters in the other array. The characters after the position defined by the `Length` field are undefined. Thus, even if the valid characters in the bounded-length strings are equal, the undefined characters are probably not equal. Figure 5-4 illustrates this problem using two bounded-length strings containing the name Mildred. Each of these bounded-length strings can hold a maximum of 20 characters. From the `Length` field we can see that each string contains seven characters. Comparing the first seven characters of each array shows that `Name_A` and `Name_B` are logically identical. Now, if we were to compare these two records with the `=` operator, *all* components in the two arrays would be compared. The `=` operator would return `False` for these records.

It is easy for a programmer using the Ada 83 implementation to slip and incorrectly use the `=` operator instead of the `Equal` function defined in the package. Ada 95 allows us to define our own `=` operator for private types thereby avoiding the possibility using the wrong operator to determine whether two bounded-length strings are equal.

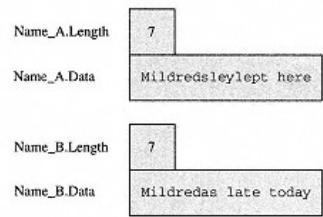


Figure 5-4  
Two "Identical" Bounded-Length Strings

### Access Types

In previous sections we defined, used, and implemented a bounded-length string class. An obvious problem with this class is the necessity to establish an upper bound on the length of string objects. Choosing too small a bound may limit an application and choosing too large a bound may waste memory. The unbounded-length string class offers a solution to this problem. But before we can implement an unbounded-length string class, we must take some time to teach you about another Ada feature: access types. *Access types* allow us to change the amount of memory used by objects while the program is running.

### Dynamic Memory Allocation

During the elaboration of an object (variable or constant), Ada finds and allocates the memory necessary for that object. We take advantage of elaboration to declare local variables whose size cannot be determined until the procedure is called. Here is an example:

```
procedure Demonstrate_Elaboration (Str : in out String) is
 Local_Str : String (Str'Range);
 .
 .
 .
```

---

**Dynamic Allocation**  
Creation of storage space in memory for a variable during run time.

---

The size of the variable `Local_Str` is not known until the procedure is called. During the elaboration of `Local_Str`, the size of the parameter `Str` is used to determine how much memory to allocate to `Local_Str`. The elaboration of the declaration in this procedure demonstrates one kind of **dynamic storage allocation**. If Ada can create dynamic variables, why can't we? The answer is, *we can*. Ada allows a program to allocate space for data dynamically, during the course of its execution.

---

**Access Type Variable**  
A variable that provides access to an object of a designated type.

---

To allocate space dynamically in Ada, we use **access type variables**. The value of an access type variable (the access value) contains *information on the location* of an unnamed object of a designated type. Exactly what constitutes an access value is not specified by Ada. An access value might be as simple as the actual address of the unnamed object. It also may include information on the object's size, number of dimensions, and so on. The details of an access value are not important. We picture an access variable as a variable that points to an object (see Figure 5-5). In fact, other languages use the term *pointer variable* instead of access variable.\*

There are three phrases commonly used to describe the association between an access variable and its object. We can say that the access variable *designates* an object, *points to* an object, or *references* an object. The process of accessing the data in an object designated by an access variable is called *dereferencing*.

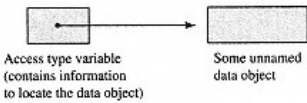


Figure 5-5  
An Access Type Variable and Its Referenced Object

---

\*The term *pointer* refers to a memory address. Because Ada's access types often contain additional information or are implemented as offsets rather than addresses, the term *pointer* is not accurate for these types.

Let's start with a very simple example. The following type declaration defines an access type that designates `Integer` objects. The declarations for variables `A` and `B` define variables that contain access values, that is, information needed to locate an `Integer` object.

```

type Integer_Ptr is access Integer; -- An access type

A : Integer_Ptr; -- The value of each of these variables provides
B : Integer_Ptr; -- access to a memory location containing an Integer

C : Integer; -- An ordinary Integer;
```

**The Null Access Value**

|                                                                                                  |
|--------------------------------------------------------------------------------------------------|
| <b>Null</b>                                                                                      |
| A literal that can be assigned to an access type variable to indicate that it points to nothing. |

For each access type, there is a literal, `null`, which is an access value designating no object at all. When access type variables such as `A` and `B` are elaborated, they are given an initial value of `null` indicating that currently the access type variable points to no object. Access types are one of only two types for which Ada automatically supplies an initial value. (The other is the task type, a type beyond the scope of this text.) Figure 5-6 illustrates the variables `A` and `B` after their elaboration.

**Using the Allocator `New`**

An *allocator* is an operation that creates space for an object during the execution of the program. The predefined allocator operation in Ada is named `new`. This allocator has two forms, which we illustrate in the following assignment statements:

```

A := new Integer; -- Allocate space for an integer
B := new Integer'(17); -- Allocate space and assign a value of 17
```

Do you remember exactly how an assignment statement is executed? First, the expression on the right side is evaluated and then the result is assigned to the variable. Evaluation of the `new` operation allocates enough memory for the object (whose type is given after the word `new`) and yields an access value that identifies the location of this memory. This access value then is assigned to the access variable on the left side of the assignment operator. Figure 5-7 illustrates the state of variables `A` and `B` after these two assignment statements have been executed. We use arrows to represent the access values stored in the two variables. Each arrow points to the object designated by the access type variable. Only the

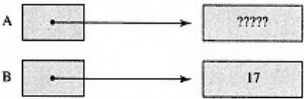
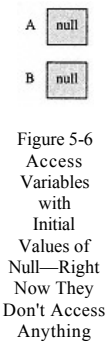


Figure 5-7  
Access Variables with Non-null Access Values

Integer object designated by variable B contains a value. This value was specified in the allocator expression. The object that variable A points to is undefined.

The syntax of an allocator is

```
allocator ::= new subtype_indication |
 new qualified_expression

qualified_expression ::= type_mark'(expression) |
 type_mark'aggregate
```

As we saw above, the value of the qualified expression is assigned as an initial value to the newly allocated memory.

### Accessing Data through Access Values

We cannot access data in storage obtained through a `new` operation by name because it doesn't have a declared name. Figure 5-7 illustrates the lack of names. There is no name given for the object with the value of 17. We access this data through an access variable that designates it. We access the object designated by an access variable, using a dot notation similar to that used to access the fields of a record variable. To access the object designated by an access variable we use the variable name followed by `.all`. As we mentioned earlier, this process is called *dereferencing*. Here are some examples of dereferencing access variables A and B:

```
Integer_IO.Put (Item => B.all);
Integer_IO.Get (Item => A.all);
C := A.all + B.all;
```

The first statement calls procedure `Put` to display the `Integer` object designated by B. In the second statement, an integer is obtained from the user and assigned to the object designated by A. The third statement assigns the sum of the two objects designated by A and B to the integer variable C. Figure 5-8 shows the results of these statements (assuming the user entered a value of 3) on the variables and designated objects. Note that A and B are access variables that designate unnamed data by location, while C is a named variable that contains integer data.

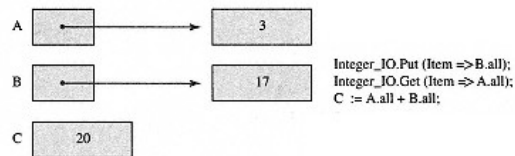


Figure 5-8  
Values After Executing the Three Statements  
(Assuming the User Enters 3)

An access variable name with `.all` refers to the object designated by the access variable. An access variable name by itself refers to the access value (location information) it contains. To emphasize the difference between access variables and the objects they designate, consider the following two assignment statements:

```
A.all := B.all;
A := B;
```

In the first assignment statement the object designated by A is assigned the value of the object designated by B. Figure 5-9 shows the results: the two objects have identical values and the two access variables have different values (they designate different objects).

| Alias                                                           |
|-----------------------------------------------------------------|
| One of several access variables that designate the same object. |

The second assignment statement assigns a *copy* of the access value (information about the *location* of B . all) stored in B to A . A and B contain identical values. What does it mean when two access variables have identical values? It means that both designate the same object. Figure 5-10 shows the results of this assignment. An access variable that designates the same object as another access variable is an **alias** of the other access variable. Figure 5-10 shows that A is an alias of B. The use of aliases can result in undesirable side effects. For example, if we assign 0 to A . all, the value of B . all is also changed to 0. Such side effects can be prevented by avoiding aliases. However, as we show later, temporary aliases are often necessary to manipulate data structures based on access variables.

### Using Unchecked\_Deallocation

Figure 5-10 illustrates another problem encountered when assigning a value to an access variable. The object that A originally designated is no longer accessible; it has no name and is not designated by any access variable. This object is lost forever! And, more importantly, the memory used by this object is lost. Programs that lose memory in this manner are said to have *memory leaks*. The Ada LRM (Language Reference Manual) allows an implementation to reclaim the memory used by objects that are no longer accessible. The reclaimed memory is then recycled for future allocation operations. This process is colorfully called *garbage collection*. However, we are not aware of any Ada system that has implemented garbage collection.

The Ada LRM also allows an implementation to supply a generic procedure called Unchecked\_Deallocation that our programs can use to explicitly recycle the memory used by unwanted objects. To make this generic procedure available, we must include the context clause with Unchecked\_Deallocation; in our program. As with any generic procedure, we use Unchecked\_Deallocation to create (via an instantiation) a procedure tailored to our needs. Here is the declaration for the generic procedure Unchecked\_Deallocation:

```
generic
 type Object is limited private;
 type Name is access Object;
procedure Unchecked_Deallocation (X : in out Name);
```

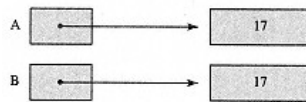


Figure 5-9  
After Executing A.all := B.all;

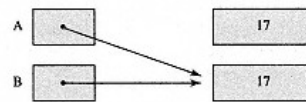


Figure 5-10  
After Executing `A := B`; the Object  
Originally Designated by A Is Lost

This generic function has two generic formal parameters: `Object` and `Name`. We have not yet seen the second of the generic formal types used here. Recall that the designation of `limited private` means that the formal type `Object` matches any type of actual parameter. Type `Name` matches any actual access type that designates type `Object`. The following instantiation uses our declarations for Integer objects given on page 194:

```
procedure Free is new Unchecked_Deallocation (Object => Integer;
 Name => Integer_Ptr);
```

As with any instantiation, we can choose any name we like; we chose to name our procedure `Free`. Procedure `Free` has a single parameter, `X`, whose type is `Integer_Ptr` (the type of the actual generic parameter associated with the formal generic parameter `Name`). Here are two example calls of this procedure:

```
Free (X => A); -- Call using named parameter association
Free (B); -- Call using positional parameter association
```

In both cases, the memory used by the object designated by the given access variable is reclaimed by the system and the access variable is assigned the value of `null`—it no longer points to any object.

### Why is it Called `Unchecked_Deallocation`?

One of the goals of the Ada language designers was that the language be suitable for writing reliable programs. Use of procedures to explicitly reclaim memory can be dangerous. This danger is a direct result of the side effects of aliases. Let's look at an example using our access variables `A` and `B` and the procedure `Free` we instantiated from `Unchecked_Deallocation`.

```
A := B; -- Make A an alias of B
Free (A); -- Recycle the memory used by the object designated by A
B.all := 0; -- Cannot say what happens
```

Figure 5-10 shows the result of executing the first assignment statement. Figure 5-11 shows the result of calling `Free (A)`. After this call to `Free`, `A` contains `null`—it no longer points to any object. `B` is not changed by the call to procedure `Free`. Its value still designates some object, although the state of this object is undefined. In the best case, attempting to use the object designated by `B` raises some exception, and in the worst case (if the memory has been allocated again) changes some other dynamic object.



Figure 5-11  
After Executing `A := z`; and `Free (A)`

The name `Unchecked_Deallocation` is a warning for anyone who uses it. This procedure does not check to see if any other access variables are aliases of the parameter designating the memory we want to reclaim. It is up to the programmer to make sure that any aliases are assigned a value of `null`.

### ***Exceptions and Access Types***

`CONSTRAINT_ERROR` is raised whenever an attempt is made to dereference an access variable that contains a null access value. Because an access variable is assigned `null` when it is elaborated, this error is commonly associated with trying to access a designated object before it has been allocated. Procedures instantiated from `Unchecked_Deallocation` do not raise `CONSTRAINT_ERROR` when passed a null access value. Here are some examples illustrating the raising of `CONSTRAINT_ERROR`:

```
Free (A); -- Recycle the memory (A is now null)
Free (A); -- CONSTRAINT_ERROR not raised here
Integer_IO.Put (Item => A.all); -- CONSTRAINT_ERROR is raised here

B := null; -- B's object now lost
Integer_IO.Put (Item => B.all); -- CONSTRAINT_ERROR is raised here
```

The exception `STORAGE_ERROR` is raised when the system runs out of memory to allocate. This exception might be raised anytime a variable is elaborated or an allocator (`new`) is used. If you see this exception, look for an infinite loop. If programs with memory leaks (remember Figure 5-10) run long enough, the available memory becomes exhausted. Such programs are difficult to debug because they may have to run for days or weeks before exhausting memory. It is best to design your program to avoid memory leaks. One way to help prevent memory leaks is to encapsulate all access variable dereferencing in packages. We use this approach for the unbounded-length string package developed later in this chapter.

### ***Access of Structured Types***

All of our previous examples of access variables designated integer objects. We used integer objects to keep our examples as simple as possible. There are no reasons why you should use access types instead of simple integer variables. The real power of access types comes when the objects they designate are structured types such as records and arrays. For example, later in this chapter we use access types that designate unconstrained arrays to implement an unbounded-length string class. Throughout the remainder of this book we use access types that designate records to implement a wide variety of classes. But first, we look at examples of the use of the allocator operation with some simple structured types.

```
subtype Name_String is String (1..20); -- A constrained array subtype
type Name_Ptr is access Name_String;
```

```
First : Name_Ptr;
Middle : Name_Ptr;
Last : Name_Ptr;
```



All three of these variables are access variables that designate objects that are arrays of 20 characters. As with all access variables, these three variables have an initial value of `null` [Figure 5-12(a)]. We use the allocator operation `new` to obtain memory for objects designated by these access variables.

```
First := new Name_String; -- No initial value
Middle := new Name_String'(1..20 => 'A'); -- An array aggregate
Last := new Name_String'("This string has 20 C"); -- A string literal
```

Figure 5-12(b) shows the results of these three assignment statements. The characters in the string designated by `First` are undefined. The string designated by `Middle` contains 20 As and the string designated by `Last` contains the string "This string has 20 C". We can operate on these string objects as we do on any string object. We can, for example, index and slice them.

```
Text_IO.Put (Item => Last.all); -- The entire 20 character string
Text_IO.Put (Item => Last.all (1..4)); -- A four character slice
Text_IO.Put (Item => Last.all(1)); -- The first character
```

```
Middle.all(1..5) := "Hello"; -- Replace the 1st 5 characters
Middle.all(6) := 'B'; -- Replace the 6th character
```

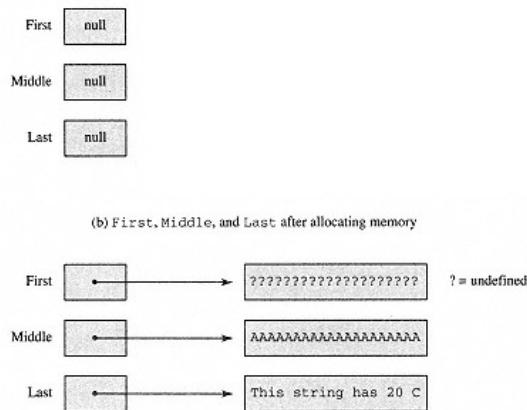


Figure 5-12  
Access Variables That Designate String Objects

When access types are used to designate arrays, Ada permits us to leave out the `.all` when dereferencing an element or slice in a designated array. Using this shorthand syntax, the statements above can be rewritten as follows:

```

Text_IO.Put (Item => Last.all); -- The entire 20 character string
Text_IO.Put (Item => Last (1..4)); -- A four character slice
Text_IO.Put (Item => Last (1)); -- The first character

Middle(1..5) := "Hello"; -- Replace the 1st 5 characters
Middle(6) := 'B'; -- Replace the 6th character

```

Because the first statement dereferences the entire array rather than a slice or individual element, we must include `.all`. You probably noticed that the shorthand syntax makes it appear as though `Last` and `Middle` are array variables rather than access type variables. In fact, some Ada programmers prefer the shorthand form because it allows them to change their implementation easily from access variables to array variables and vice versa. We believe that such decisions should be made during program design rather than during program implementation. Some programmers also prefer the shorthand syntax because it makes it easier to enter their programs from the keyboard. We believe that the extra documentation provided by `.all` is well worth the extra time it takes to type the four characters. Because of the extra documentation provided, most of the examples in this text use the `.all` form. You (or your professor) can decide which form is best for your programs.

We can instantiate a procedure from `Unchecked_Deallocation` that can be used to recycle the 20 character string objects designated by variables of type `Name_Ptr`. Here is a code fragment that creates an instance of `Unchecked_Deallocation` and frees the memory designated by `First`:

```

procedure Free is new Unchecked_Deallocation (Object => Name_String;
 Name => Name_Ptr);

.
.
.

Free (First); -- Reclaim the memory designated by First

```

As a general practice we take advantage of Ada's overloading abilities and use the name `Free` for every procedure that we instantiate from `Unchecked_Deallocation`. Because each one has a different access type for a parameter, the Ada compiler is able to determine which `Free` procedure to use.

### An Unbounded-Length String Class

Access types also may designate *unconstrained* arrays and records. We use this capability to implement an unbounded-length string class.

```

type Unbounded_String_Type is access String;

W : Unbounded_String_Type;
X : Unbounded_String_Type;
Y : Unbounded_String_Type;
Z : Unbounded_String_Type;

```

Each of the four access variables declared above can designate a string of any size. The size of the designated object is established by the allocator operator `new`. The size can be

specified as a range constraint on the subtype indication (as in the first line below) or through an initial value (as in the last three lines below).

```
W := new String(1..20); -- String of 20 characters, no initial value
X := new String'("Hello"); -- String of 5 characters with initial value
Y := new String'(1..10 => 'J'); -- String of 10 J's
Z := new String'(X.all & ' ' & "Mildred"); -- "Hello Mildred"
```

Figure 5-13 shows the result of these four allocation operations.

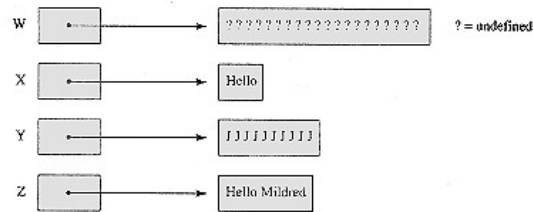


Figure 5-13  
Access Variables That Designate Unconstrained String Objects

The algorithms for implementing the operations for unbounded-length strings are very similar to those we used for bounded-length strings earlier in this chapter. However, the potential for aliases and memory leaks presents additional problems. For example, we think of the assignment operator as one that creates copies of objects. This concept certainly is valid for every data type we have encountered before access types. As we saw earlier, normal assignment with access types creates an alias rather than a copy. For example, the assignment statement

```
W := X; -- Attempt to assign a copy of string X to string W
```

makes W an alias of X. Both access variables now designate the same object, the string containing "Hello". This assignment also leaks memory. The string of 20 characters originally designated by W can no longer be accessed. Figure 5-14 shows the results of this assignment statement. To make W a copy of X, we first must reclaim the memory designated by W and then allocate new memory for the copy. The following two statements accomplish this "assignment":

```
Free (W); -- Reclaim memory
W := new String'(X.all); -- Allocate a new object with an initial value
 -- equal to the string designated by X
```

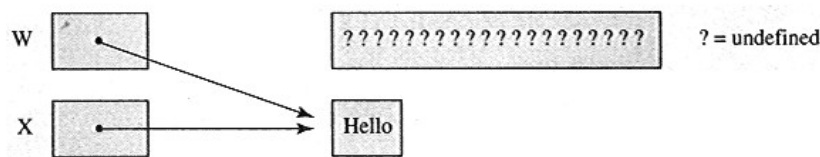


Figure 5-14  
After Executing `W := X;`, W and X Are Aliases and Memory Is Lost

Figure 5-15 shows the results of this two-statement "assignment."

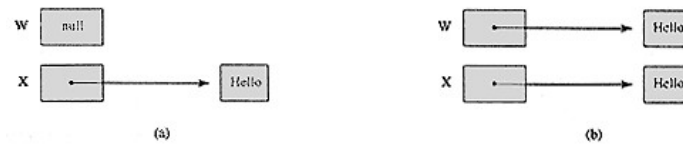


Figure 5-15  
After Executing (a) `Free (W);` and (b) `W := new String'(X.all);`

The assignment operator is not the only construct that creates aliases and memory leaks. Passing access type parameters creates aliases because it is location information that is passed, not the object. Using access types requires a good deal of care. As a result, access types are not permitted in most safety-critical software (programs whose failure might result in death, injury, loss of equipment, or environmental harm). This is not to say that access types should never be used. Through packages and private types, Ada allows us to encapsulate all access variable dereferencing. By limiting the manipulation of access based data structures to a few well-tested packages, we can markedly reduce the inherent risks of access types.

### The Abstract Level

Specification 5-2 defines our unbounded-length string class. There are two big differences between this specification and the one for bounded-length strings (Specification 5-1). First, we have made our string type `limited private`. By limiting our type, the user of the class is limited to the operations defined in our package; an application programmer cannot use the assignment operator to create aliases or memory leaks. Our specification includes an operation called `Assign` to carry out assignment correctly. Second, our package uses procedures instead of functions for all operations that return a string (assign, concatenate, slice, and conversion). This change was also made to prevent an application programmer from creating aliases and memory leaks (see Exercise 11). Notice the use of `in out` mode for the result parameters in these procedures. Each procedure recycles the "old" memory designated by the result before assigning the answer to it.

### Specification 5-2— Unbounded-Length Strings

```
with Text_IO;
package Unbounded_Strings is

-- This package implements an unbounded-length string type.

 type Unbounded_String is limited private; -- Unbounded-length string type

 Null_Unbounded_String : constant Unbounded_String; -- A string with no characters

 -- Relational operators
```

```

function "<" (Left : in Unbounded_String; Right : in Unbounded_String) return Boolean;
function "<=" (Left : in Unbounded_String; Right : in Unbounded_String) return Boolean;
function ">" (Left : in Unbounded_String; Right : in Unbounded_String) return Boolean;
function ">=" (Left : in Unbounded_String; Right : in Unbounded_String) return Boolean;
function "=" (Left : in Unbounded_String; Right : in Unbounded_String) return Boolean;

```

```

-- Catenation operators

```

```

procedure Catenate (Left : in Unbounded_String;
 Right : in Unbounded_String;
 Result : in out Unbounded_String);
procedure Catenate (Left : in Unbounded_String;
 Right : in Character;
 Result : in out Unbounded_String);
procedure Catenate (Left : in Character;
 Right : in Unbounded_String;
 Result : in out Unbounded_String);
procedure Catenate (Left : in Unbounded_String;
 Right : in String;
 Result : in out Unbounded_String);
procedure Catenate (Left : in String;
 Right : in Unbounded_String;
 Result : in out Unbounded_String);

```

```

procedure Assign (Target : in out Unbounded_String;
 Source : in Unbounded_String);
-- This procedure is used as an "assignment statement" for unbounded-length strings.
--
-- Preconditions: None
--
-- Postconditions: The string Target is a copy of the string Source.

```

```

function Length (Source : in Unbounded_String) return Natural;
-- This procedure returns the number of characters in a string.
--
-- Preconditions: None
--
-- Postconditions: The number of characters in Source is returned.

```

```

procedure Slice (Source : in Unbounded_String;
 Low : in Positive;
 High : in Natural;
 Result : in out Unbounded_String);
-- This procedure returns a substring of the given string.
--
-- Preconditions: None
--
-- Postconditions: If Low <= High, a string composed of the characters from
-- position Low to position High is returned.
-- Otherwise, a null string is returned.

```

```

--
-- Exceptions: CONSTRAINT_ERROR is raised if Low <= High and
-- High is greater than Length (Source).

function Element (Source : in Unbounded_String;
 Index : in Positive) return Character;
-- This function returns the character in the string at the position given by Index.
--
-- Preconditions: None
--
-- Postconditions: The character in Source at Position is returned.
--
-- Exceptions: CONSTRAINT_ERROR is raised if Position is
-- greater than Length (Source).

function Index (Source : in Unbounded_String;
 Pattern : in Unbounded_String) return Natural;
-- This procedure searches for Pattern in Source.
--
-- Preconditions: None
--
-- Postconditions: If the Pattern is found within Source, Index returns the smallest
-- index I such that the slice of Source starting at I matches
-- Pattern.
-- Otherwise, 0 is returned.

-- Conversions

function To_String (Source : in Unbounded_String) return String;
-- This function converts an unbounded-length string to a fixed-length string.
--
-- Preconditions: None
--
-- Postconditions: The characters in Source are returned as
-- an array of characters.

procedure To_Unbounded_String (Source : in String;
 Target : in out Unbounded_String);
-- This procedure converts a fixed-length string to an unbounded-length string.
--
-- Preconditions: None
--
-- Postconditions: The characters in Source are returned
-- as a bounded-length string.

-- Input and Output Operations

```

```

procedure Get_Line (File : in Text_IO.File_Type := Text_IO.Current_Input;
 Item : in out Unbounded_String);
-- Unbounded-length string equivalent to Text_IO.Get_Line

procedure Put (File : in Text_IO.File_Type := Text_IO.Current_Output;
 Item : in Unbounded_String);
-- Unbounded-length string equivalent to Text_IO.Put

procedure Finalize (Object : in out Unbounded_String);
-- Release all of the memory associated with an unbounded string
-- This procedure should be called for a local variable before returning.
--
-- Preconditions: None
--
-- Postconditions: The memory used by Object is released. Object is undefined.

private

 type Unbounded_String is access String;

 Null_Unbounded_String : constant Unbounded_String := new String'("");

end Unbounded_Strings;

```

There are some other differences between this specification and the one we presented for bounded-length strings. We implemented the bounded-length string class as a generic package to allow the user to set the maximum string length. As there is no maximum length for unbounded-length strings, we implement this class without any need for using generics.

This package declaration also contains a procedure called `Finalize`. To prevent memory leaks, `Finalize` must be called by the client using the package for all local unbounded-length string variables before control leaves the scope of those variables. As you know, the memory for local variables is deallocated when control leaves their scope. If these local variables are access variables, the memory for the access variable is deallocated but the memory used by any objects the local access variables designate is not reclaimed.\* The `Finalize` procedure is the only hint in the visible part of the package specification that makes the user aware that access types are being used: *all* dereferencing is done within the package body.

### ***The Application Level***

The use of procedure `Finalize` is illustrated in Program 5-2, an unbounded-length string version of our application to sort names. `Finalize` is called to reclaim the memory used by the local variables in procedures `Selection_Sort` and `Last_Name_First`. The other changes we made in this version of the program were to substitute calls to procedure `Assign` where we used the `:=` operator in our bounded-length string version and to substitute procedure calls for function calls to comply with the unbounded-length specification.

---

\*Ada 95 provides automatic reclamation of dynamically allocated memory through its controlled types.

**Program 5-2—****Sort Names (Unbounded-Length Strings)**

```

with Unbounded_Strings;
use Unbounded_Strings;
with Text_IO;
procedure Sort_Names is

-- This program reads names from the keyboard and displays them in alphabetical
-- order. Names may be entered one per line in one of two forms:
--
-- First_Name Last_Name or Last_Name, First_Name
--
-- The comma is required in the second form.
-- End of input is indicated with a null name.
--
-- Assumptions: 1. There are no more than 20 names.
-- 2. All input lines contain less than 30 characters.
-- 3. There is exactly one blank between the first and last names.
-- 4. There are no leading or trailing blanks.
-- 5. There are no blanks within a first name or a last name.

Max_Names : constant := 20; -- Maximum number of names

-- Array type for holding all the names
type Index_Type is range 0..Max_Names;
subtype Index_Range is Index_Type range 1..Max_Names;
type Name_Array is array (Index_Range range <>) of Unbounded_String;

Name : Unbounded_String; -- One name
Name_List : Name_Array (Index_Range); -- List of names entered by the user
Num_Names : Index_Type; -- Number of names entered

procedure Selection_Sort (List : in out Name_Array) is

-- An unordered list is taken as input. The same list
-- with the components in ascending order is returned.

Name : Unbounded_String; -- Temporary variable for swapping
Min_Index : Index_Type; -- Index of minimum so far

begin -- Selection_Sort
 for Pass_Count in List'First .. List'Last - 1 loop
 -- Find the index of the smallest item in List(Pass_Count)..List(List'Last)
 Min_Index := Pass_Count;
 for Place_Count in Pass_Count + 1 .. List'Last loop
 -- Compare names with Unbounded_Strings."<"
 if List(Place_Count) < List(Min_Index) then
 Min_Index := Place_Count;
 end if;
 end loop;
 end loop;
end Selection_Sort;

```



```

-- Swap List(Min_Index) and List(Pass_Count)
Assign (Target => Name, Source => List(Min_Index));
Assign (Target => List(Min_Index), Source => List(Pass_Count));
Assign (Target => List(Pass_Count), Source => Name);
end loop;
Finalize (Name); -- Release memory for local variable Name
end Selection_Sort;

```

```

procedure Last_Name_First (Name : in out Unbounded_string) is
-- This procedure converts a name from the form
-- First_Name Last_Name to Last_Name, First_Name.
-- If the name is already last name first, it is not changed.
--
-- Preconditions: There is exactly one blank between the first and last
-- names. If last name is first, it is followed by a
-- comma.

```

```

Position : Natural; -- Position result of string search
Last_Name : Unbounded_String;
First_Name : Unbounded_String;
Blank : Unbounded_String;

```

```

begin
-- Determine the position of the blank
To_Unbounded_String (Source => " ", Target => Blank);
Position := Index (Source => Name, Pattern => Blank);
-- Is the first name first (no comma indicates it is)?
if Element (Name, Position-1) /= ',' then -- Change the order
-- Get the last name
Slice (Source => Name, Low => Position+1,
 High => Length(Name), Result => Last_Name);
-- Get the first name
Slice (Source => Name, Low => 1,
 High => Position-1, Result => First_Name);
-- Add a comma and a blank after the last name
Catenate (Left => Last_Name, Right => ", ", Result => Last_Name);
-- Combine the last and first name
Catenate (Left => Last_Name, Right => First_Name, Result => Name);
end if;
Finalize (Last_Name); -- Release memory
Finalize (First_Name); -- of all
Finalize (Blank); -- local variables
end Last_Name_First;

```

```

begin -- Sort_Names
-- Get all the names
Num_Names := 0;
Input_Loop: -- Each iteration, one name is processed.
loop
Text_IO.Put_Line ("Enter a name (blank name to end)");
Unbounded_Strings.Get_Line (Item => Name);

```

```
-- Is this a null name?
exit Input_Loop when Length (Name) = 0;
-- Make sure the last name comes before the first name
Last_Name_First (Name);
-- Put the name into the list
Num_Names := Num_Names + 1;
Assign (Target => Name_List (Num_Names) , Source => Name);
end loop Input_Loop;

-- Sort the names
Selection_Sort (List => Name_List (1..Num_Names));

-- Display the sorted names
for Index in 1..Num_Names loop
 Unbounded_Strings.Put (Item => Name_List (Index));
 Text_IO.New_Line;
end loop;

end Sort_Names;
```

It is important to note that except for the `Finalize` operation, the user of this package doesn't need to be aware that the strings are in dynamically allocated memory. Even though this program uses access variables such as `Name`, the details of the access operations are completely hidden in the package body.

This program is considerably longer and more complicated than the earlier one that used bounded-length strings. Making the unbounded-length string type limited private is responsible for the additional complications. However, the alternative is worse. Permitting a client to use the assignment operator for this type is sure to lead to programs with hard to find bugs that are a direct result of inadvertent aliases and memory leaks. Encapsulating these details helps us create safer programs.

### *The Implementation Level*

Body 5-2 contains the implementation of package `Unbounded_Strings`. The algorithms are very similar to the bounded-length operations given earlier. Let's look at the operations that are different.

**Body 5-2—**

## Unbounded-Length Strings Implements Specification 5-2

```
with Unchecked_Deallocation; -- Generic Procedure for reclaiming memory
package body Unbounded_Strings is
-- This package implements a unbounded-length string type.
```

[illegible]

```

-- Relational operators

```

```
function "<" (Left : in Unbounded_String; Right : in Unbounded_String)
 return Boolean is
begin
 return Left.all < Right.all;
end "<";
```

```
function "<=" (Left : in Unbounded_String; Right : in Unbounded_String)
 return Boolean is
begin
 return Left.all <= Right.all;
end "<=";
```

```
function ">" (Left : in Unbounded_String; Right : in Unbounded_String)
 return Boolean is
begin
 return Left.all > Right.all;
end ">";
```

```
function ">=" (Left : in Unbounded_String; Right : in Unbounded_String)
 return Boolean is
begin
 return Left.all >= Right.all;
end ">=";
```

```
function "=" (Left : in Unbounded_String; Right : in Unbounded_String)
 return Boolean is
begin
 return Left.all = Right.all;
end "=";
```

```

-- Catenation operators

```

```
procedure Catenate (Left : in Unbounded_String;
 Right : in Unbounded_String;
 Result : in out Unbounded_String) is
 Local_Result : Unbounded_String;
begin
 Local_Result := new String'(Left.all & Right.all); -- Combine the two source strings
 Free (Result); -- Free memory currently designated by Result
 Result := Local_Result; -- Set Result to designate the combined strings
end Catenate;
```

```
procedure Catenate (Left : in Unbounded_String;
 Right : in Character;
 Result : in out Unbounded_String) is
 Local_Result : Unbounded_String;
```

```

begin
 Local_Result := new String'(Left.all & Right); -- Combine the two source strings
 Free (Result); -- Free memory currently designated by Result
 Result := Local_Result; -- Set Result to designate the combined strings
end Catenate;

procedure Catenate (Left : in Character;
 Right : in Unbounded_String;
 Result : in out Unbounded_String) is
 Local_Result : Unbounded_String;
begin
 Local_Result := new String'(Left & Right.all); -- Combine the two source strings
 Free (Result); -- Free memory currently designated by Result
 Result := Local_Result; -- Set Result to designate the combined strings
end Catenate;

procedure Catenate (Left : in Unbounded_String;
 Right : in String;
 Result : in out Unbounded_String) is
 Local_Result : Unbounded_String;
begin
 Local_Result := new String'(Left.all & Right); -- Combine the two source strings
 Free (Result); -- Free memory currently designated by Result
 Result := Local_Result; -- Set Result to designate the combined strings
end Catenate;

procedure Catenate (Left : in String;
 Right : in Unbounded_String;
 Result : in out Unbounded_String) is
 Local_Result : Unbounded_String;
begin
 Local_Result := new String'(Left & Right.all); -- Combine the two source strings
 Free (Result); -- Free memory currently designated by Result
 Result := Local_Result; -- Set Result to designate the combined strings
end Catenate;

procedure Finalize (Object : in out Unbounded_String) is
begin
 Free (Object);
end Finalize;

procedure Assign (Target : in out Unbounded_String;
 Source : in Unbounded_String) is
-- This procedure must work when Source and Target designate the same string.
 Result : Unbounded_String;
begin
 Result := new String'(Source.all); -- Make a copy of the Source string
 Free (Target); -- Free memory currently designated by Target
 Target := Result; -- Set Target to designate the copy
end Assign;

```

```

procedure Slice (Source : in Unbounded_String;
 Low : in Positive;
 High : in Natural;
 Result : in out Unbounded_String) is
 Local_Result : Unbounded_String;
begin
 if High >= Low then
 Local_Result := new String'(Source.all(Low..High));
 else
 Local_Result := new String'(""); -- Null range, return null string
 end if;
 Free (Result); -- Free memory currently designated by Result
 Result := Local_Result; -- Set Result to designate the substring
end Slice;

function Length (Source : in Unbounded_String) return Natural is
begin
 return Source.all'Length;
end Length;

function Element (Source : in Unbounded_String;
 Index : in Positive) return Character is
begin
 return Source.all(Index);
end Element;

function Index (Source : in Unbounded_String;
 Pattern : in Unbounded_String) return Natural is

 First : Positive; -- Pattern is compared to
 Last : Positive; -- Source (First..Last)

begin -- Search
 First := 1; -- Begin search at the first position of Source
 Last := Pattern.all'Length;
 Search_Loop:
 loop -- Exit when there are fewer characters remaining in Source to check
 -- than there are in the Pattern or when match is found.
 exit when Last > Source.all'Length or else
 Source.all(First..Last) = Pattern.all;
 First := First + 1;
 Last := Last + 1;
 end loop Search_Loop;
 if Last <= Source.all'Length then -- Did we find the pattern?
 return First;
 else
 return 0;
 end if;
end Index;

```

```

function To_String (Source : in Unbounded_String) return String is
begin
 return Source.all; -- Return the array
end To_String;

procedure To_Unbounded_String (Source : in String;
 Target : in out Unbounded_String) is
begin
 Free (Target); -- Free memory currently designated by Target
 Target := new String'(Source); -- Set Target to designate a copy of Source
end To_Unbounded_String;

-- Input and Output Operations

procedure Get_Line (File : in Text_IO.File_Type := Text_IO.Current_Input;
 Item : in out Unbounded_String) is
 Char : Character;
begin
 Free (Item); -- Free memory currently designated by Item
 Item := new String'(""); -- Initialize result
 -- Each iteration get one character and concatenate it to the end of the string
 while not Text_IO.End_Of_Line (File) loop
 Text_IO.Get (File => File, Item => Char);
 Catenate (Left => Item, Right => Char, Result => Item);
 end loop;
 Text_IO.Skip_Line (File);
end Get_Line;

procedure Put (File : in Text_IO.File_Type := Text_IO.Current_Output;
 Item : in Unbounded_String) is
begin
 Text_IO.Put (File => File, Item => Item.all);
end Put;

end Unbounded_Strings;

```

Procedure `Assign` contains one more step than the algorithm we discussed earlier. A local copy of the source string is made before the memory used by the target string is recycled. We made this change to handle the possibility of the source and target being the same string (equivalent to `Apple := Apple`). Although we never intentionally assign a variable to itself, we often write algorithms (such as the swap portion of a sort algorithm) that in certain circumstances do just that. Using the following code fragment, let's trace a call to procedure `Assign` to see exactly how access values are passed as parameters:

```

X : Unbounded_Strings.Unbounded_String;
Y : Unbounded_Strings.Unbounded_String;
.
.
.

```

```
Assign (Target => X, Source => Y); -- X becomes a "copy" of Y
```

Let's say that at the time of the call to `Assign`, `X` designates the string "Jose" and `Y` designates the string "Mildred". Figure 5-16(a) shows the relevant variables prior to the procedure call. Note that before the call to `Assign`, no memory has been allocated for the parameters `Source` and `Target` or the local variable `Result`.

When procedure `Assign` is called, memory is allocated for the parameters and local variable of procedure `Assign`, the value of `X` is copied into `Target`, and the value of `Y` is copied into `Source`. Because `X` and `Y` contain access values, copying them make `Target` and `Source` aliases as shown in Figure 5-16(b).

Execution of the assignment statement

```
Result := new String'(Source.all);
```

allocates new memory for a string object. This object is initialized to the value "Mildred". The location of this new object is assigned to the variable `Result`. Figure 5-16(c) shows the result of executing this assignment statement.

The call to `Free (Target)` deallocates the memory designated by `Target` and assigns `null` to `Target`. However, the alias `X` is unchanged; it now designates an undefined object. We take care of this problem later. Figure 5-16(d) shows the result of the call to `Free`. Execution of the assignment statement

```
Target := Result;
```

makes `Target` an alias of `Result`. Figure 5-16(e) shows the results of this assignment statement.

When control is returned to the caller of procedure `Assign`, the value of the formal parameter `Target` is copied to the actual parameter `X`. As Figure 5-16(f) shows, `X` now designates the string "Mildred" which is a copy of the string designated by `Y`. We have accomplished what we set out to do. As always, the memory used by the parameters and local variables is deallocated when control leaves the procedure.

You should spend some time studying the other operations in this body. Be sure to draw pictures of the access variables and their objects as you trace through these operations.

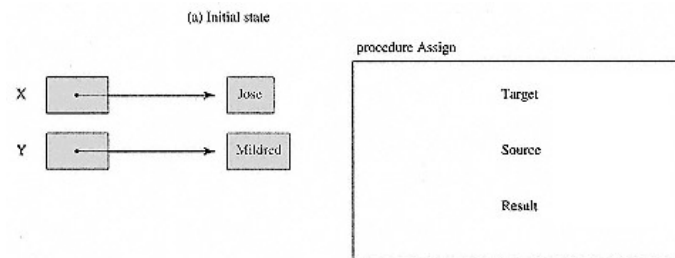
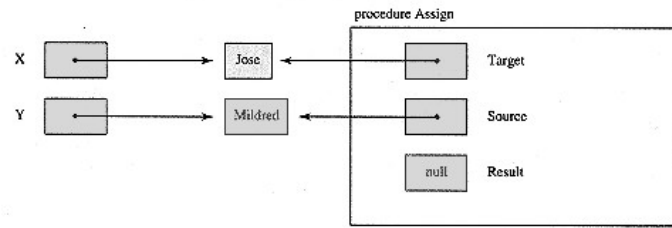
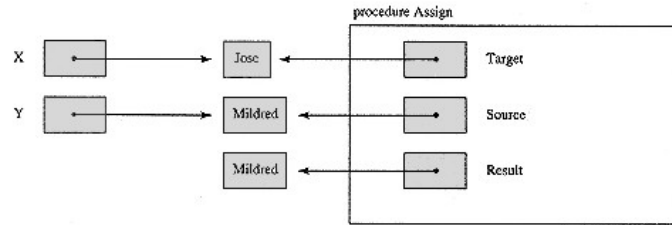


Figure 5-16  
Trace of a Call to Procedure Assign

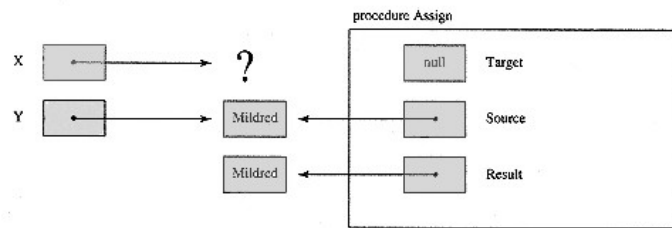
(b) Procedure Assign has just been called



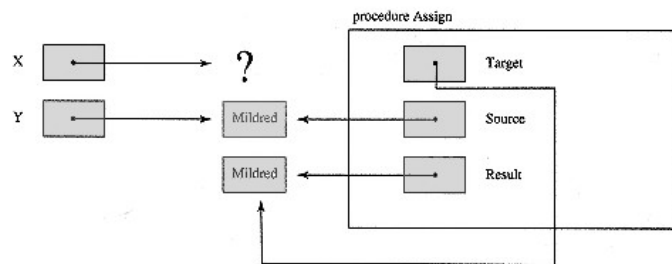
(c) After executing `Result := new String'(Source.all);`



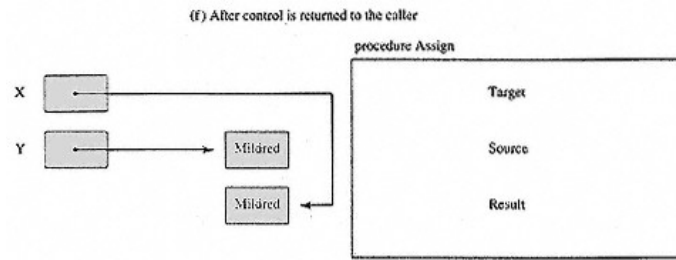
(d) After executing `Free (Target);`



(e) After executing `Target := Result;`







## Organization of Memory

You may have noticed that we did not draw the memory for the bottom string object containing the word "Mildred" in the box enclosing the variables for procedure `Assign`. Where does the allocator actually find the memory requested by our program? When a program is running, its memory space is divided into a number of regions. The actual code (the machine language version of our Ada statements) is stored in one region. Named variables (the ones we declare) are allocated memory in a region called the *stack*. When we use the allocator `new` to obtain memory for an object, the system allocates memory in a third region usually called the *heap*. Figure 5-17 illustrates the memory used by a typical program.

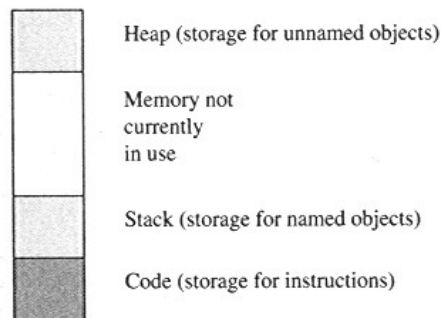


Figure 5-17  
Memory Used by a Running Program

When our program calls a subprogram, memory for its parameters and local variables is allocated on the stack. In Figure 5-17, the boundary between the stack and the unused memory moves upward to add this memory to the stack. When control is returned, this memory is deallocated by moving the boundary back down. When our Ada program uses the allocator `new` to obtain memory, the run-time system allocates it from the heap. This memory may come from recycled memory somewhere in the middle of the heap or from extending the heap into the unused memory region. In Figure 5-17, the boundary between the heap and unused memory moves downward. When the unused memory region becomes exhausted and our program requests additional memory (from the stack or heap), `STORAGE_ERROR` is raised.

## Ada 95 Implementation Note

A programmer using our unbounded-length string package may forget to include calls to procedure `Finalize` to reclaim the memory used by objects designated by local access variables. Any resulting memory leaks may go unnoticed until the program crashes with a `STORAGE_ERROR` exception after running for a long time.

Having to use the procedure `Assign` rather than the `:=` operator is enforced by our limited private type declaration. Although this guarantees that assignment is done by creating a copy rather than an alias, it forces the user to use an inconvenient syntax.

Both of these problems are solved by Ada 95's controlled types. These types allow the package writer to redefine the `:=` operator so that a copy is made rather than an alias. Controlled types also allow the package programmer to specify what happens to objects designated by local access variables when these local variables are deallocated.

Finally, we note that Ada 95 provides an unbounded-length string class (predefined package `Ada.Strings.Unbounded`) with about 35 operations. As you would expect, a user of this class need not call a finalize procedure to recycle the memory designated by local variables.

## Summary

Strings are used extensively to manipulate text. There are three different string classes: fixed-length strings, bounded-length strings, and unbounded-length strings. Both Ada 83 and Ada 95 provide type `String` to implement fixed-length strings. The operations available in Ada 83 for this string class are limited to the primitive array operations of slicing, indexing, and catenation. Ada 95 supplies additional operations for fixed-length strings. Ada 95 also provides a bounded-length string class and an unbounded-length string class. In the chapter we develop implementations of bounded-length strings and unbounded-length strings.

Access types allow a programmer to allocate memory for objects while the program is executing. All access variables are initialized with the null access value. A null access value indicates that the access variable currently designates no object. Ada's allocator operation `new` is used to allocate memory for an object. The allocation may include an initial value. Once memory for an object has been allocated it is accessed through an access variable.

Access types are most useful when they designate structured types. We implemented an unbounded-length string class using access types that designate unconstrained arrays of characters (type `String`). When designing classes that use access types, draw "box and arrow" pictures to help you implement the operations.

The two major problems with using access variables are aliasing and memory leakage. An alias is an access variable that designates the same object as another access variable. Memory leaks occur when the only access variable designating an object is changed without reclaiming the memory used by the object. Calling a procedure instantiated from the generic procedure `Unchecked_Deallocation` reclaims the memory.

You should encapsulate all classes implemented with access types (as we did the unbounded-length string class) so that anyone using them have little awareness that access types are used. We can eliminate aliases and dramatically reduce the possibility of memory leaks by limiting the type and implementing all operations as procedures. In Ada 83 we cannot write a class that guarantees no memory leaks. The application programmer must explicitly call a procedure (like `Unbounded_Strings.Finalize`) to reclaim memory used by local variables. Ada 95 provides controlled types that we can use to prevent all memory leaks.

## Exercises

1. (a) What is a fixed-length string?  
 (b) What is a varying-length string?  
 (c) What is a bounded-length string?  
 (d) What is an unbounded-length string?
2. (a) Why would you choose to use a bounded-length string class rather than an unbounded-length string class?  
 (b) Why would you choose to use an unbounded-length string class rather than a bounded-length string class?
3. Complete the following function (using the packages instantiated on page 179) that converts a `String132` string to a `String40` string. If `Value` contains over 40 characters, truncate it to 40 characters.

```
function To_String40 (Value : in String132.Bounded_String)
 return String40.Bounded_String is
```

4. What is the difference between an access object (variable) and a designated object?
5. What is the initial value that Ada assigns to all access variables when they are elaborated?
6. You are given the following declarations:

```
subtype Name_Type is String (1..10);
subtype GPA_Type is Float range 0.0..4.0;
type Student_Rec is
 record
 First_Name : Name_Type;
 Last_Name : Name_Type;
 ID : Positive;
 GPA : GPA_Type;
 Current_Hours : Natural;
 Total_Hours : Natural;
 end record;

type List_Type is array (1..100) of Student_Rec;

type List_Ptr is access List_Type;

Student_List : List_Ptr;
```

Assume that an integer or character takes one cell in memory, a float number or access value takes two cells, and that the fields are in contiguous memory locations with no gaps.

- (a) How much space (number of cells) is allocated to the variable `Student_List` when it is elaborated?
- (b) How much space is allocated at run time by the statement

```
Student_List := new List_Type;
```

Assume that `new` allocates exactly the number of "cells" needed by the data type.

- (c) Write an assignment statement to set the ID of the first student in the list array to 1000.
- (d) Write a loop to print out the IDs of all the students in the list. Assume that all array positions are in use.
- (e) Write a loop to add the `Current_Hours` field of each student to the `Total_Hours` field, and to reset the `Current_Hours` to zero.

(f) Write the necessary declaration to instantiate a procedure called `Free` that reclaims the memory of an object of an access variable of type `List_Ptr`.

(g) Use the procedure `Free` you created in part (f) to reclaim the memory designated by `Student_List`. What is the value of the variable `Student_List` after calling procedure `Free`?

7. You are given the following declarations:

```
type Frequency_Array is array (Character range <>) of Natural;
```

```
type Frequency_Ptr is access Frequency_Array;
```

```
A : Frequency_Ptr;
```

```
B : Frequency_Ptr;
```

```
C : Frequency_Ptr;
```

(a) Write a statement that allocates space for an array with an index range of 'a'..'g' that is designated by access variable A. Do not give initial values to the elements in the array.

(b) Write a loop to assign the value of 100 to all elements in the array designated by access variable B.

(c) Write a statement that allocates space for an array with an index range of 'h'..'n' that is designated by access variable B. Each element in the array should be given an initial value of zero.

(d) Draw a picture to illustrate the value of B after the statement you wrote for part (c) is executed.

(e) Write a statement that allocates space for an array with an index range of 'a'..'n' that is designated by access variable C. Each element in the array should be given an initial value corresponding to the values in the arrays designated by access variables A and B.

(f) Write the necessary declaration to instantiate a procedure called `Free` that reclaims the memory of an object of an access variable of type `Frequency_Ptr`.

(g) Use the procedure `Free` you created in part (e) to reclaim the memory designated by C.

8. Using the declarations given in the previous question and the procedure `Free` you instantiated in part (f) of that question, which of the following code fragments raises `CONSTRAINT_ERROR`? Which results in memory leakage? Assume that before each code fragment is executed, the three access variables A, B, and C designate different objects.

(a) `Free(A);`  
`Free(A);`

(b) `Free(A);`  
`A.all('c') := 52;`

(c) `A := null;`

(d) `A := null;`  
`A.all('c') := 52;`

(e) `A := B;`  
`B.all('f') := 96;`

(f) `A := B;`  
`Free(B);`  
`A := null;`

(g) `A.all := C.all('a'..'g');`

```
(h) A.all := C.all
```

```
(i) A := null;
 B := A;
```

9. Draw pictures like those used in Figure 5-16 to trace the following call of procedure `Assign`:

```
Unbounded_Strings.Assign (Target => X, Source => X);
```

10. The major dangers of using access types to implement a class are the creation of aliases and memory leaks.

(a) Explain how using limited private types to implement a class prevents the application programmer from creating aliases and memory leaks.

(b) Even when using limited private types, it is still possible for the application programmer to create a memory leak. Explain how this is possible. Hint: why is procedure `Unbounded_Strings.Finalize` needed? Ada 95 provides controlled types that eliminate the need for a finalize operation.

11. All operations in package `Unbounded_Strings` that return strings are written as procedures instead of functions. Let's see what happens when we use a function instead of a procedure. Here is a function to concatenate two unbounded-length strings:

```
function "&" (Left : in Unbounded_String;
 Right : in Unbounded_String)
 return Unbounded_String is
 Result : Unbounded_String; -- The combined strings
begin
 -- Create a new string that is the combination of Left and Right
 Result := new String'(Left.all & Right.all);
 return Result;
end "&"
```

And here are declarations of four unbounded-length string variables and the code an application programmer could write to assign values to them:

```
A : Unbounded_Strings.Unbounded_String;
B : Unbounded_Strings.Unbounded_String;
C : Unbounded_Strings.Unbounded_String;
D : Unbounded_Strings.Unbounded_String;
.
.
.
Unbounded_Strings.To_Unbounded_String (Source => "Mildred",
 Target => A);
Unbounded_Strings.To_Unbounded_String (Source => "R. ";
 Target => B);
Unbounded_Strings.To_Unbounded_String (Source => "Smedley");
 Target => C);
Unbounded_Strings.To_Unbounded_String (Source => "Beasley");
 Target => D);
```

(a) Draw pictures like those used in Figure 5-16 to demonstrate that the following code fragment does *not* produce a memory leak.

```
Unbounded_Strings.Assign (Target => D,
 Source => A & C);
```

(b) Draw pictures like those used in Figure 5-16 to demonstrate that the following code fragment *does* produce a memory leak.

```
Unbounded_Strings.Assign (Target => D,
 Source => A & B & C);
```

## Programming Problems

1. (a) Add comments for preconditions, postconditions, and exceptions to the following operations to be added to the bounded-length string package developed in this chapter.

```
function "*" (Left : in Natural; Right : in Character)
 return Bounded_String;
-- This function returns a string comprised of Left copies
-- of the character in Right.

function "*" (Left : in Natural; Right : in Bounded_String)
 return Bounded_String;
-- This function returns a string comprised of Left copies
-- of the string in Right.

function Delete (Source : in Bounded_String;
 From : in Positive;
 Through : in Natural) return Bounded_String;
-- This function returns a string comprised of Source with
-- the characters between position From to position Through
-- deleted. If Through < From, Source is returned unchanged.

function Trim (Source: in Bounded_String) return Bounded_String;
-- This function returns a string obtained by removing from Source
-- all leading and trailing blanks.
```

(b) Implement the operations.

(c) Write a test plan to test your operations, and write a driver program to carry out your test plan.

2. Solve problem 1 using the unbounded-length string package instead of the bounded-length string package.

3. Write an Ada program to read the text of a newspaper story in a data file (input lines are contain no more than 80 characters) and produce an output file with lines containing no more than 32 characters on them and no more than 30 lines on a page. Each line should contain as many words as possible. No word is to be split between lines. There should be exactly one blank between words in the output file. There may be many blanks between words in the input file. Use the bounded-length string class in your solution. It is useful to preinstantiate an instance so that all of your program units can use the class.

This problem can be solved more easily if we consider the input file to be a file of words instead of the normal file of characters. You should write a package to implement this new file class. Operations could include `Open`, `Close`, `End_Of_File`, `Get`, and `Unget`. `Get` returns a word, `End_Of_File` returns `True` when there are no more *words* in the file object, and `Unget` returns a word to the file object. `Unget` can be called when you find that there is not enough room on the current line for the word you just got. The next call to `Get` returns the word you saved by calling `Unget`.

**6—****Stacks****GOALS**

After reading this chapter, you should be able to

- describe a stack and its operations at an abstract level
- determine when a stack is an appropriate data structure for a specific problem
- design and implement the solution to a problem for which a stack is an appropriate data structure
- evaluate a postfix expressions
- show how stacks can be used to evaluate postfix expressions
- use a static array to implement a stack class
- use a linked structure to implement a stack class
- explain the difference between static and dynamic allocation of the space in which a stack is stored
- compare stack implementations in terms of source code length, use of storage space, and Big-O approximations of the stack operations
- choose between a static and a dynamic linked stack structure

In Chapter 4 we looked at the set, an *unordered* collection of items. In this chapter and the next chapter we examine two data structures (the stack and the queue) that are collections of items *ordered by time* of their entry into the structure.

## The Abstract Level

### What Is a Stack?

#### Stack

A data structure in which elements are added and removed from only one end; a "last in, first out" (LIFO) structure.

What is a stack? Consider the items pictured in Figure 6-1. Although the objects are all different, each illustrates a common concept: the **stack**.

At the abstract level, a stack is an ordered group of elements. The removal of existing elements and the addition of new elements can take place only at the *top* of the stack. For instance, if your favorite blue shirt is underneath a faded, old red one in a stack of shirts, you first must remove the red shirt (the top element) from the stack. Only then can you remove the desired blue shirt, which is now the top element in the stack. Then you may replace the red shirt on the top of the stack or throw it away.

The stack is considered an "ordered" group of items because elements occur in a sequence according to how long they've been in the stack. The items that have been in the stack the longest are at the bottom; the most recent are at the top. At any time, given any two elements in a stack, one is higher than the other. (For instance, the red shirt was higher in the stack than the blue shirt.)

Because items are added and removed only from the top of the stack, the last element to be added is the first to be removed. There is a handy mnemonic to help you remember this rule of stack behavior: A stack is a *last in, first out* (LIFO) structure.

The accessing operation of a stack is summarized as follows: both to retrieve elements and to assign new elements, access only the top of the stack.

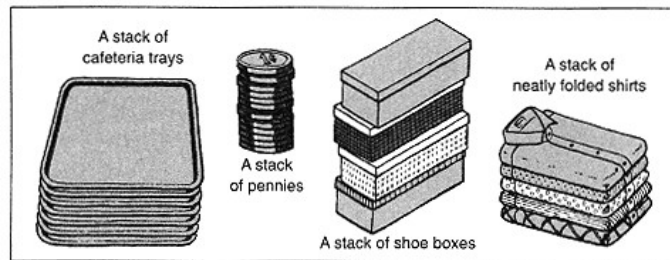


Figure 6-1  
Real-Life Stacks

### Operations on Stacks

The logical picture of the structure is only half the definition of a class. The other half is a set of operations that allows the user to access and manipulate the elements stored in the structure. Given the abstract structure of a stack, what kinds of operations do we need in order to use a stack?



A stack is a dynamic structure—it changes as elements are added to and removed from it. When we begin using a stack, it should be empty (containing no elements). The operation that adds an element to the top of a stack is usually called *Push*, and the operation that takes the top element off the stack is referred to as *Pop*. We also must be able to tell whether a stack contains any elements before we pop it, so we need a Boolean operation *Empty*. As a logical data structure, a stack is never conceptually "full," but for a particular implementation you may need to test whether a stack is full before pushing. We'll call this Boolean operation *Full*. We also might want an operation that destroys a stack, getting rid of all the elements left in it and leaving the stack empty. We'll call this operation *Clear*. Figure 6-2 shows how a stack, envisioned as a stack of building blocks, is modified by several Push and Pop operations.

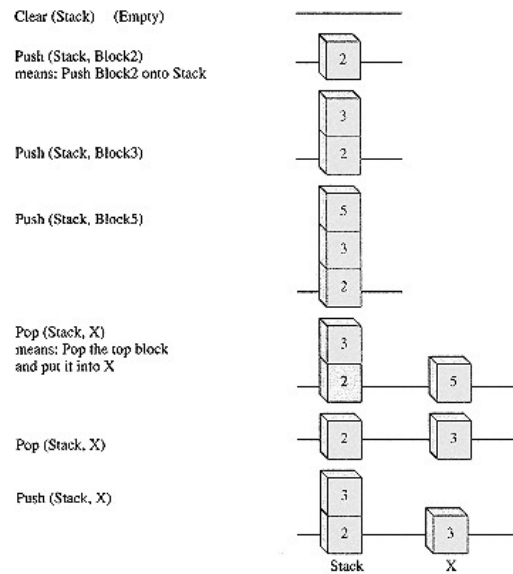


Figure 6-2  
The Effects of Push and Pop Operations

Next we must formalize the stack operations we have described. As with our set and string classes, we use a package specification to describe our class accurately. We'll make this a generic package so that our stack abstraction can be easily reused in different applications. We then can instantiate stack packages for stacks containing any type of objects. Our generic stack specification is given in Specification 6-1.

## Specification 6-1— Stack Class

generic

```
type Element_Type is private; -- The stack element
```

package Stack is

```
-- This package implements a stack, a data structure in which elements are added
-- and removed from only one end; a "last in, first out" (LIFO) structure.
```

```
type Stack_Type (Max_Size : Positive) is limited private; -- The stack class
```

```
UNDERFLOW : exception;
OVERFLOW : exception;
```

```

procedure Clear (Stack : in out Stack_Type);
-- Remove all elements from the stack
```

```
-- Preconditions : None
--
-- Postconditions : Stack is empty; it contains no elements
```

```

function Empty (Stack : in Stack_Type) return Boolean;
-- Tests whether a stack is empty (contains no elements)
```

```
-- Preconditions : None
--
-- Postconditions : Empty = (stack is empty)
```

```

function Full (Stack : in Stack_Type) return Boolean;
-- Tests whether a stack is full (no more elements can be pushed on it)
```

```
-- Preconditions : None
--
-- Postconditions : Full = (stack is full)
```

```

procedure Push (Stack : in out Stack_Type;
 New_Element : in Element_Type);
-- Adds New_Element to the top of Stack
```

```
-- Preconditions : None
--
-- Postconditions : Stack = original Stack with New_Element added on top
--
-- Exceptions : OVERFLOW Raised on attempt to Push a new element onto a
-- full stack. Stack is unchanged.
```

```

procedure Pop (Stack : in out Stack_Type;
 Popped_Element : out Element_Type);
-- Removes top element from Stack and returns it in Popped_Element

-- Preconditions : None
--
-- Postconditions : Stack = original Stack with top element removed
-- Popped_Element = top element of original Stack
--
-- Exceptions : UNDERFLOW Raised on attempt to Pop an element from an
-- empty stack. Stack remains empty.

private

-- We'll fill this in later.

end Stack;

```

### Exceptions

---

#### Stack Underflow

The condition resulting from trying to pop an empty stack.

#### Stack Overflow

The condition resulting from trying to push an element onto a full stack.

---

In addition to the stack operations we discussed in the last section, this package contains the declarations for two exceptions. If a stack is empty when we try to pop an element from it, the resulting error condition is called **stack underflow**. The exception comments for procedure `Pop` indicate that the exception `UNDERFLOW` is raised if this condition is detected. **Stack overflow** is the condition resulting from trying to push a new element onto a stack that is already full. Procedure `Push` raises the exception `OVERFLOW` if this condition is detected.

Why bother declaring these exceptions in the stack package? Why not just display an error message? For example, instead of raising `UNDERFLOW` when the `Pop` operation detects an erroneous condition, `Pop` could display an error message on the console such as

**Error! The stack is empty; you can't remove an item at this time.**

One problem of this approach is that the error message we choose when we write the stack package body probably has little meaning in the context of the client program that uses a stack. Imagine the confusion if the above message were displayed when someone running a word processing program clicks their mouse on the Undo command. A more meaningful error message in this situation would be "There are no more commands to undo."

Part of the task of writing a reusable component is to allow errors that are detected by the component to be dealt with at the *appropriate level of abstraction*. Ada's exceptions are the perfect tool for this job. When the stack package detects a problem, it raises an exception that is propagated back to the client program to deal with. Exception propagation also ensures that the error condition is not ignored. If the client has no handler for the exception, it is propagated back to the operating system which ends execution of the program and displays a (usually cryptic) error message.

### Exceptions and Postconditions

Postconditions are assertions that state what results are to be expected at the exit of an operation or procedure. Our style of commenting

postconditions for package operations is to divide the exit assertions into two groups. The assertions in the group labeled *Postconditions* are true if the operation terminates normally; that is, when no exception is raised. The comment section labeled *Exceptions* contains assertions that are true if the package operation terminates abnormally by raising an exception. Notice how our exception comments specify exactly what the state of the stack is after the package raises the exception. Other people prefer to have just a single postcondition comment section that includes the exit assertions found in both of our comment sections.

### The Application Level

Now let's look at an example of how the operations in the stack package might be used in a program. Since you were in elementary school you have written arithmetic expressions using a format known as *infix* notation. This same notation is used for writing arithmetic expressions in Ada. The operator in an infix expression is written in between its operands. When an expression contains multiple operations such as

$$5 + 2 \times 6$$

we need to use a set of rules to determine which operation to carry out first. You learned in your mathematics classes that multiplication is done before addition. You learned Ada's operator precedence rules in your first programming class. In both situations, we use parentheses to override the normal ordering rules. It is easy to make a mistake writing or interpreting an infix expression containing multiple nested sets of parentheses.

### Evaluating Postfix Expressions

*Postfix* notation is another format for writing arithmetic expressions. In this notation, the operator is written after the two operands. Here are some simple postfix expressions and their results:

| <i>Postfix Expression</i> | <i>Result</i> |
|---------------------------|---------------|
| 4 5 +                     | 9             |
| 9 3 /                     | 3             |
| 17 8 -                    | 9             |

The rules for evaluating postfix expressions with multiple operators are much simpler than those for evaluating infix expressions: simply evaluate the operators from left to right. There are no precedence rules to learn and parentheses are never needed. Because of this simplicity, some popular hand-held calculators use postfix notation to avoid the complications of the multiple parentheses required in non-trivial infix expressions (Figure 6-3).

Let's look at a postfix expression containing two operators:

$$6 \ 2 \ / \ 5 \ +$$

We evaluate the expression by scanning from left to right. The first item, 6, is an operand, so we go on. The second item, 2, is also an operand, so again we continue. The third item

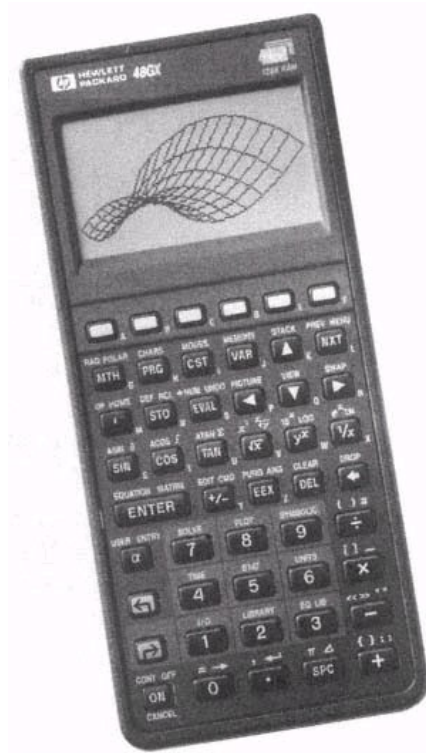


Figure 6-3  
A Calculator That Evaluates Postfix  
Expressions

is the division operator. We now apply this operator to the two previous operands. Which of the two saved operands is the divisor? The one we saw most recently. We divide 6 by 2 and substitute 3 back into the expression replacing  $6/2$ . Our expression now looks like this:

$3\ 5\ +$

We continue our scanning. The next item is an operand, 5, so we go on. The next (and last) item is the operator  $+$ . We apply this operator to the two previous operands giving a result of 8.

Here's another example:

$4\ 5\ +\ 7\ 2\ -\ *$

Scanning from left to right, the first operator encountered is  $+$ . This is applied to the two preceding operators giving us the expression

$9\ 7\ 2\ -\ *$

The next operator encountered is  $-$ , so we subtract 2 from 7 giving us

$9\ 5\ *$

Finally, the last operator,  $*$ , is applied to its two preceding operands giving our final answer of 45.

Here are some more examples of postfix expressions containing multiple operators and the results of evaluating them. See if you get the same results when you evaluate them.

| <i>Postfix Expression</i> | <i>Result</i> |
|---------------------------|---------------|
| 4 5 7 2 + - *             | -16           |
| 3 4 + 2 * 7 /             | 2             |
| 5 7 + 6 2 - *             | 48            |
| 4 2 3 5 1 - * + *         | 56            |
| 4 2 + 3 5 1 - * +         | 18            |

Now suppose we need to write a program that evaluates postfix expressions entered interactively from the keyboard. We can use the same approach as we did by hand. We get items one at a time from the line. When the item we get is an operator, we apply it to the last two operands. Because we can't back up on the input line, we must save previously scanned operands. A stack is the ideal place to store the previous operands because the top item is always the most recent operand and the next item on the stack is always the second most recent operand—just the two operands required when we find an operator. Here is an algorithm that uses a stack in this manner to evaluate a postfix expression entered from the keyboard:

### Postfix Evaluation

```

loop
 exit when no more items on line
 Get Item
 if Item is an operand
 Push Item onto the stack
 else
 Pop last two values from the stack
 Perform the operation with the two values
 Push the result onto the stack
 end if
end loop
Pop the stack to obtain the final result

```

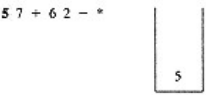
Each iteration of this loop processes one operator or one operand from the input line. When an operand is found, there is nothing to do with it (we haven't yet found the operator to apply to it) so we save it on the stack until later. When an operator is found, we get the two operands from the stack, do the operation, and put the result back on the stack; the result may be an operand for a future operator.

Let's trace this algorithm. Before we enter the loop, the input that has not yet been processed and the stack look like this:

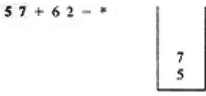
5 7 + 6 2 - \*



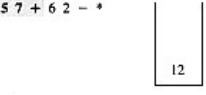
After one iteration of the loop, we have processed the first operand and pushed it onto the stack.



After the second iteration of the loop, the stack contains two operands.



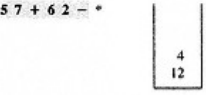
The + operator is encountered in the third iteration. We pop the two operands from the stack, perform the operation, and push the result onto the stack.



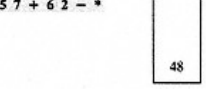
In the next two iterations of the loop, two operands are pushed onto the stack.



When the - operator is found, the top two operands are popped, subtracted, and the result pushed onto the stack.



When the \* operator is found, the top two operands are popped, multiplied, and the result pushed onto the stack.



Now that we have processed all of the items on the input line, we exit the loop. The final result, 48, is popped from the stack.

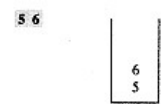


Invalid Expressions

Every valid postfix expression (1) must have exactly one more operand than it has operators and (2) must have two operands preceding each operator. Here are some examples of invalid postfix expressions that break one of these two rules:

| <i>Invalid Postfix Expression</i> | <i>Reason Why Invalid</i>      |
|-----------------------------------|--------------------------------|
| 5 6                               | Too many operands              |
| 5 6 * 7                           | Too many operands              |
| / 6 2                             | / not preceded by two operands |
| 6 / 2                             | / not preceded by two operands |
| 6 2 / * 4                         | * not preceded by two operands |

What happens if the user enters an invalid postfix expression? Let's trace our algorithm using the first invalid expression in the list above. In the first loop iteration, 5 is pushed onto the stack. Then 6 is pushed on during the second iteration. The stack now looks like this:



Because there are no more items on the input line, we exit the loop and pop the final result. Here is what things look like at the end of the algorithm trace:



Although the algorithm is complete, a number remains on the stack. When we trace any postfix expression with excess operands we always find one or more values remaining on the stack when our algorithm is complete. So anytime we find that the stack is not empty when our algorithm is complete, we know that the expression is invalid.



The second example on the invalid expression list also has too many operands (three operands and only one operator). Try tracing our algorithm with this invalid postfix expression. Is there still an operand on the stack when you reach the end of the algorithm?

Let's trace the third example in our list of invalid expressions. After determining that `/` is an operator, we try to pop two values from the stack. Because there is nothing on the stack, the first call to `Pop` raises `UNDERFLOW`. In the fourth example, our algorithm first pushes 6 onto the stack. Then it finds the `/` and attempts to pop its two operators from the stack. The second call to `Pop` raises `UNDERFLOW`. In our last example, `Pop` raises `UNDERFLOW` when it is called to get the second operand for `*`. Anytime that `UNDERFLOW` is raised, our postfix expression must be invalid.

Is `OVERFLOW` ever raised by our postfix evaluation algorithm? This exception is raised when we attempt to push another value onto a full stack. Our algorithm pushes operands onto the stack until an operator is encountered. `OVERFLOW` is raised when there are more operators preceding an operand than there is space in the stack. The valid postfix expression

```
1 2 3 4 5 6 + + - + *
```

would thus overflow a stack that holds only five elements.

Now we're ready to implement this algorithm in Ada. To simplify our input processing, we assume that the operands entered are unsigned single digits (0 to 9) and that there are no blanks typed after the last item on the line. The first assumption allows us to process both operators and operands as single characters. The second assumption allows us to use the `Text_IO.End_Of_Line` function to determine easily when all the items in the input line have been processed. Programming problem 1 asks you to implement the algorithm without these simplifying assumptions. So that we can evaluate many postfix expressions, we enclose our postfix algorithm in a loop. When the result of an expression is 0, we exit the loop. (Is an expression consisting of only 0 a valid postfix expression?)

Although the operands entered by the user must be unsigned single digits, the results and intermediate results can be any integer value. Because these results must be stored on our stack, we choose to use a stack of integers.

We already know how to detect invalid postfix expressions. We also handle expressions containing invalid characters (characters that are not digits or operators), those with too many operands, and those with results beyond the range of type `Integer`. Our complete program is shown in Program 6-1.

#### **Program 6-1— Postfix Evaluation**

```
with Integer_IO;
with Stack;
with Text_IO;
procedure Postfix is

-- This program accepts a postfix expression, evaluates it, and displays the result.
-- Appropriate messages are displayed for invalid expressions.
```

```

-- Assumptions: 1. All operands are unsigned single digits.
-- 2. There are no spaces after the last operator on the line.

subtype Digit is Character range '0' .. '9';

BAD_OPERATOR : exception; -- Raised when invalid operator is found

package Integer_Stack is new Stack (Element_Type => Integer);

Item : Character; -- An item (digit, blank, or operator) from the input line
Result : Integer; -- Result of evaluating an expression
Left : Integer; -- First operand
Right : Integer; -- Second operand

-- The stack for saving intermediate results
Stack : Integer_Stack.Stack_Type (Max_Size => 50);

procedure Get_Item (Item : out Character) is
-- This procedure returns the next non-blank character entered by the user.
-- Preconditions : There are characters remaining on the input line.
-- Postconditions : Item is not a blank.
 Value : Character;
begin
 Blank_Loop: -- Each iteration, one input character is processed.
 loop
 Text_IO.Get (Value);
 exit Blank_Loop when Value /= ' ';
 end loop Blank_Loop;
 Item := Value;
end Get_Item;

function To_Integer (Item : in Digit) return Integer is
-- Converts a character digit to an integer;
-- Preconditions : None
-- Postconditions : 0 <= To_Integer <= 9
begin
 return Digit'Pos (Item) - Digit'Pos ('0');
end To_Integer;

function Evaluate (Left : in Integer;
 Operator : in Character;
 Right : in Integer) return Integer is
-- Applies the given operator to the two operands
-- Preconditions : None
-- Postconditions : Evaluate = Left Operator Right
-- Exceptions : BAD_OPERATOR is raised if Operator is not +, -, *, or/
begin
 case Operator is
 when '+' =>
 return Left + Right;

```

```

when '-' =>
 return Left - Right;
when '*' =>
 return Left * Right;
when '/' =>
 return Left / Right;
when others =>
 raise BAD_OPERATOR;
end case;
end Evaluate;

```

```

begin -- Program Postfix
 Input_Loop:
 loop -- Each iteration, one expression is evaluated
 Integer_Stack.Clear (Stack);
 Text_IO.Put_Line ("Enter a postfix expression.");

 Exception_Block: -- Used to associate handlers with the code in the evaluation loop
 begin
 Evaluate_Loop:
 loop -- Each iteration, one item on the input line is processed
 exit Evaluate_Loop when Text_IO.End_Of_Line;
 Get_Item (Item);
 if Item in Digit then -- Is the Item an operand?
 -- Convert the character Item to an integer and push on the stack
 Integer_Stack.Push (Stack => Stack,
 New_Element => To_Integer (Item));
 else -- Item must be an operator
 Integer_Stack.Pop (Stack => Stack,
 Popped_Element => Right);
 Integer_Stack.Pop (Stack => Stack,
 Popped_Element => Left);
 Result := Evaluate (Left, Item, Right); -- Carry out the operation
 Integer_Stack.Push (New_Element => Result, -- Put result back on
 Stack => Stack); -- the stack
 end if;
 end loop Evaluate_Loop;
 Integer_Stack.Pop (Stack => Stack,
 Popped_Element => Result);
 if Integer_Stack.Empty (Stack) then
 Text_IO.Put ("The result is ");
 Integer_IO.Put (Result);
 Text_IO.New_Line;
 else
 Text_IO.Put_Line ("Not a valid postfix expression.");
 Result := Integer'Last; -- Assign value to loop control variable
 end if;

 exception
 when Integer_Stack.UNDERFLOW =>
 Text_IO.Put_Line ("Not a valid postfix expression.");
 Result := Integer'Last; -- Assign value to loop control variable

```

```

when Integer_Stack.OVERFLOW =>
 Text_IO.Put_Line ("Expression has too many operands for this program.");
 Result := Integer'Last; -- Assign value to loop control variable
when BAD_OPERATOR =>
 Text_IO.Put_Line ("Expression contains an invalid character.");
 Result := Integer'Last; -- Assign value to loop control variable
when CONSTRAINT_ERROR =>
 Text_IO.Put_Line ("An intermediate result is too large or too small.");
 Result := Integer'Last; -- Assign value to loop control variable
end Exception_Block;

exit Input_Loop when Result = 0;
Text_IO.Skip_Line; -- Move reading marker to next line
end loop Input_Loop;
end Postfix;

```

The exception handlers in this program help make it robust. There is only one error that is not handled through exceptions—the program explicitly tests for an empty stack after the evaluation loop.

Instead of using the `UNDERFLOW` exception, we could have tested for an empty stack before each call to `Pop`. Such mixing of algorithm steps and error handling steps makes a program more difficult to understand and therefore more difficult to debug and modify. Exception handlers allow us to separate the algorithm code from the error checking code.

### ***Other Stack Applications***

You have just seen an application in which stacks are used to evaluate a postfix arithmetic expression. Stacks also are used extensively in the evaluation of the infix arithmetic expressions you write in your Ada programs. The example of program `Postfix` hints at the types of applications that use a stack. A stack is the appropriate data structure when information must be saved and then retrieved in reverse order. A situation requiring you to backtrack to some earlier position may be a good one in which to use a stack. For instance, in trying to find the way out of a maze, you may end up against a wall and need to backtrack to another exit. If you use a stack to save the alternative paths as you pass them, you can retrace your route to an earlier position.

Have you ever wondered how a program determines where to continue executing when it gets to the end of a procedure or function? Many systems use a stack to keep track of the return addresses, parameter values (or their addresses), and other information used by subprograms. For example, when Procedure A is called, its calling information is pushed onto a "run-time stack." Then when Procedure B is called from A, B's calling information is pushed onto the top of the stack. B then calls Procedure C, and C's calling information is pushed onto the stack. When C finishes executing, the stack is popped to retrieve the information needed to return to Procedure B. Then B finishes executing, and its calling information is popped from the stack. Finally Procedure A completes, and the stack is popped again to return to the main program. Because it can grow and shrink throughout execution, according to the level of subprogram nesting, a stack is a good structure for storing data on the order of procedure calls within a program. We return to this topic in Chapter 11 when we discuss recursion.

## The Implementation Level

We now consider the implementation of our stack class. We first look at how a stack can be stored within an array. Then we see how stacks can be stored in dynamically allocated memory.

### *The Implementation of a Stack As a Static Array*

Because all the elements of a stack are of the same type, an array seems like a reasonable structure to contain them. We can put elements in sequential slots in the array, placing the first element pushed in the first array position, the second element pushed in the second array position, and so on. The floating "high-water" mark is the top element in the stack.

Before we can implement our stack as an array, we need to know how to find the top element when we want to Pop and where to put the new element when we Push. Remember that, although we can access any element of an array directly, we have agreed to use the accessing function "last in, first out" for a stack. So we access the stack elements only through the top, not through the bottom or the middle. Recognizing this distinction from the start is important. Even though the implementation of the stack may be a random-access structure such as an array, the stack itself as a logical entity is not randomly accessed. We can use only its top element.

One way to keep track of the top position in the array containing stack elements is to declare another variable, `Top`. However, the specifications of the stack operations only show the entity `Stack` passed in and out, not two entities, `Stack` and `Top`. However, we can bundle the array and the top indicator into a single entity by using a record.

In order to declare the array variable that contains the stack, we must decide on its maximum size. We could declare a constant in our stack package to set this size. This, however, isn't very flexible. The user of our stack class should be able to determine the maximum size of the stack. One way to do so is to include the maximum size as a generic parameter. Users then could supply a value for the maximum stack size when they instantiate a stack package from our generic template. This solution still imposes some limitations on our users. All of the stack objects they declare will have the same maximum size. We have used a better way, an unconstrained array type and a record with a discriminant. The users must supply a value for the record discriminant in the declaration of each different stack object. This value is used to set the size of the array of elements. Here then are the declarations for the private part of our stack package (Specification 6-1, page 224).

```
private

 type Stack_Array is array (Positive range <>) of Element_Type;

 type Stack_Type (Max_Size : Positive) is
 record
 Top : Natural := 0; -- Initialize to empty stack
 Elements : Stack_Array (1..Max_Size);
 end record;

end Stack;
```

When we include a discriminant in the complete declaration in the private section of the package, we also must include it in the incomplete declaration in the visible part of the package. We included `Max_Size` in the private type declaration in Specification 6-1.

Figure 6-4 illustrates the data structure resulting from the following declarations:

```
-- Package for a character stack class
package Char_Stack is new Stack (Element_Type => Character);

-- One stack of characters
Stack : Char_Stack.Stack_Type (Max_Size => 100);
```

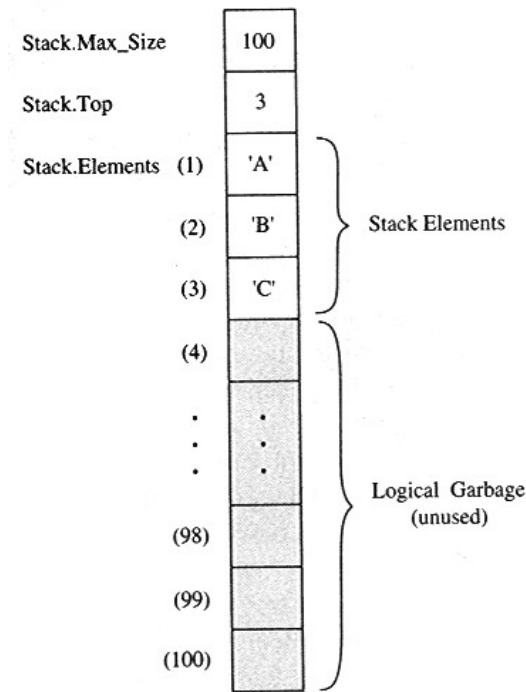


Figure 6-4  
The Array Implementation of a Stack  
Containing Three Elements—'C' Is the  
Top Element

In this data structure, `Top` is the index of the top element in the stack, not the index of the next free slot in the array.

Finally, notice that in our declaration of `Stack_Type` we have supplied an initial value of zero for the field `Top` in our stack record type. This value ensures that all stack objects declared are initially empty. It isn't necessary to blank out the whole array; instead, we simply set the top indicator to 0, as pictured in Figure 6-5. It doesn't matter how much garbage is in the array. If the stack's top indicator says that the stack is empty, none of the array slots can be accessed.

**Array-Based Stack Package Body**

Body 6-1 contains the stack-based function and procedure bodies that implement the stack operations defined in Specification 6-1. All of these operations are quite short. Procedure `Clear` simply sets the top indicator back to zero. As a zero value for `Top` indicates an empty stack, function `Empty` tests to see whether or not the top indicator is zero.

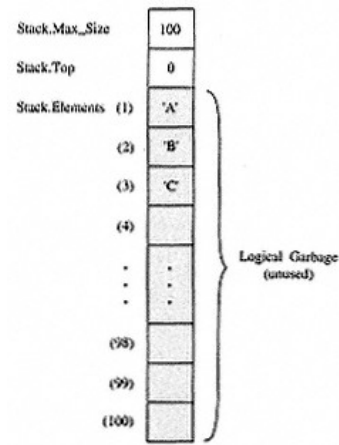


Figure 6-5  
An Empty Stack

### Body 6-1— Array-Based Stack Implements Specification 6-1

package body Stack is

```

procedure Clear (Stack : in out Stack_Type) is
begin
 Stack.Top := 0;
end Clear;
```

```

function Empty (Stack : in Stack_Type) return Boolean is
begin
 return Stack.Top = 0;
end Empty;
```

```

function Full (Stack : in Stack_Type) return Boolean is
begin
 return Stack.Top = Stack.Max_Size;
end Full;
```

```

procedure Push (Stack : in out Stack_Type;
 New_Element : in Element_Type) is
begin
 if not Full (Stack) then
 Stack.Top := Stack.Top + 1;
 Stack.Elements (Stack.Top) := New_Element;
 else
 raise OVERFLOW;
 end if;
end Push;

procedure Pop (Stack : in out Stack_Type;
 Popped_Element : out Element_Type) is
begin
 if not Empty (Stack) then
 Popped_Element := Stack.Elements (Stack.Top);
 Stack.Top := Stack.Top - 1;
 else
 raise UNDERFLOW;
 end if;
end Pop;
end Stack;

```

Recall that a stack as an abstract data structure cannot be full; however, a particular implementation may make a test for a full stack necessary. The array implementation of a stack requires the programmer to choose an upper limit for the size of each stack via a value supplied for the discriminant. Our package uses the value of the discriminant to declare the size of the array. Therefore, we must add an extra operation to the stack class to check for a full stack. To make this test, we compare the top indicator to the stack record discriminant `Max_Size`. Recall that record discriminants can be accessed like any other field in a record.

To add, or push, an element onto the top of the stack is a two-step task. We first increment the top indicator and then assign the new element to the location indicated by the top indicator. First, however, we must check for the possibility that there is no room left in the stack for the new element. We can use the `Full` operation for this check. If the stack is not already full, we assign the new element to the proper location in the array. If, however, the stack is full, we raise the exception `OVERFLOW`. Because we have no exception handler associated with procedure `Push`, this exception is propagated back to the caller of `Push`. As specified in the exception comments in our stack package declaration, the stack is not changed by the raising of this exception.

Let's take a look at the effect of a push operation on the stack in Figure 6-6(a). We want to `Push (Stack => Stack, New_Element => 'L')`. To do so we need to increment the top indicator from 3 to 4 and then put our new element, 'L', into the fourth element place. The result is shown in Figure 6-6(b).

To remove, or pop, an element from the stack, we perform the reverse of the push operation. First we check for stack underflow. If the stack is not empty, we assign the top element to our out parameter and then we decrement the top indicator. Figure 6-7 shows how the pop operation would affect a stack. In Figure 6-7(a), the value in `Stack.Top` tells us



that the top element is stored in `Stack(3)`. We assign the value of the top element, 'Y', to `Popped_Element`. Then the top indicator is decremented, giving us the stack shown in Figure 6-7(b). Note that after popping, 'Y' still is stored in the third element slot in the array, but we cannot access it through the stack. The 'Y' is now logical garbage, because the stack contains only two elements.

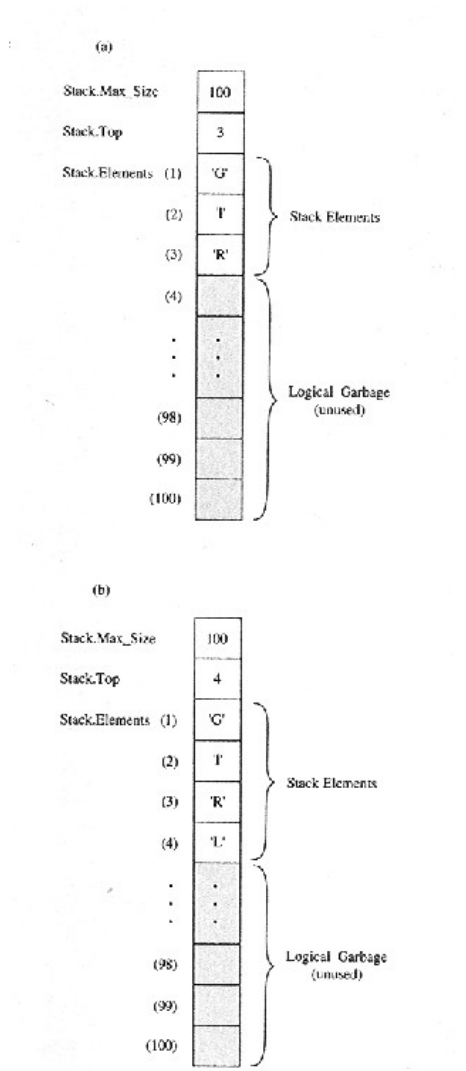


Figure 6-6  
The Effect of a Push Operation

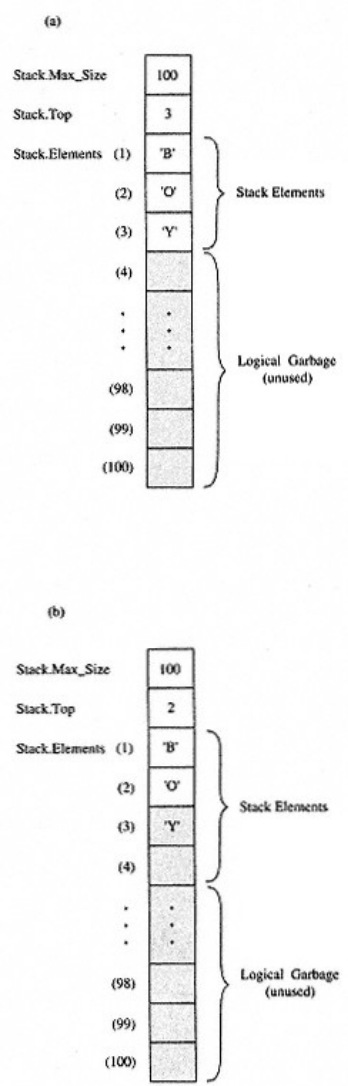


Figure 6-7  
The Effect of a Pop Operation

### The Implementation of a Stack As a Linked Structure

The implementation of a stack in an array is very simple, but it has a serious drawback: The amount of storage space for a stack must be determined at elaboration time. When we declare a variable of `Stack_Type`, memory is allocated to contain the number of stack elements specified by the discriminant. If we use fewer elements at run time, space is wasted; if we need to push more elements than the array can hold, we cannot. It would be nice if we could just get space for stack elements as we need it.

Chapter 5 introduced the concept of dynamic storage allocation—the ability to allocate memory for the program's data at run time. Let's see how we might use this concept to build a stack.

#### Implementing Procedure Push

We can modify procedure `Push` to allocate space for each new element dynamically. One seemingly apparent way to do so is to implement a stack object as an access type variable.

```
type Stack_Type is access Element_Type;
```

```
Stack : Stack_Type;
```

Figure 6-8 illustrates what such an empty stack would look like. When we want to push a new element onto our stack, we use the `new` allocator to obtain some memory for one element.

```
Stack := new Element_Type'(New_Element);
```



Figure 6-8  
An  
Empty  
Stack

Figure 6-9 illustrates how this stack would look after pushing a value of 'E'. Although everything looks fine so far, there is a major flaw in this implementation. Determine what happens after you push three more elements onto this stack using the same algorithm. Space is allocated for each new element, and the stack is assigned a value that designates the latest element. However, because our access variable can only designate a single object, access to the earlier three elements is lost. Figure 6-10 illustrates the situation.

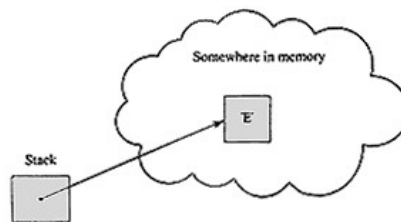


Figure 6-9  
After Pushing the First Element, 'E', onto the Stack

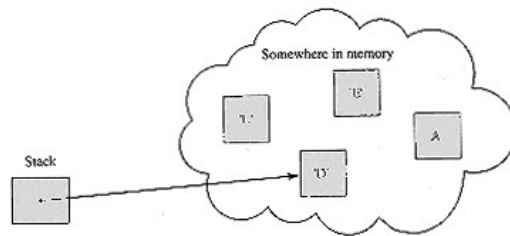


Figure 6-10  
After Pushing 'E', 'L', 'A', and 'D' onto the Stack

Somehow we must have an access value stored for each element in our stack. One possibility that comes to mind is to declare our stack as an array of access types and to have each element of this array designate an element in the stack, as shown in Figure 6-11. This solution would keep track of the pointers to all the elements in the correct order, but it wouldn't solve our original problem: We still need to declare an array of a particular size. Where else can we put the access values?

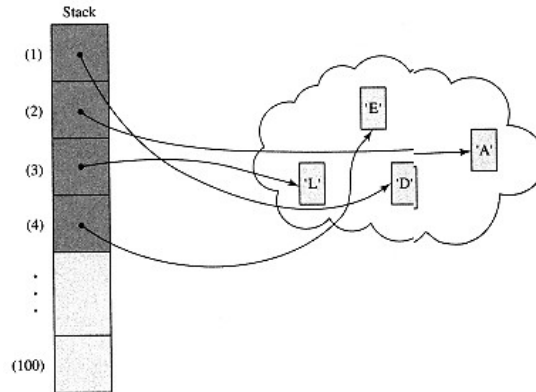


Figure 6-11  
One Way to Keep Track of the Access Values

---

**Node**

A record containing data and an access value that designates another node.

---

We need to be able to allocate memory to store each new stack element *and* allocate memory to store a new access value. We can accomplish this task by allocating a record containing both the new stack element and an access value. Records containing both data and an access value are called **nodes**. We use the access value in the node to designate the next node in the stack. Figure 6-12 illustrates a single node, a record containing the fields `Info` and `Next`. Figure 6-13 illustrates how nodes can be linked together to make a stack containing the values 'A', 'L', and 'E'. In this stack, 'A' is on the top and 'E' is on the bottom.

`Stack` is an access variable that designates the top node in the stack. The top node contains an access value that designates the node "underneath." The Ada syntax for dereferencing a record object is exactly the same as it is for dereferencing any object: the access

variable name, followed by `.all`. To dereference a *particular field* of the record, we use normal record syntax (dot notation) after the `.all`. The following examples illustrate the dereferencing of the nodes in the stack as shown in Figure 6-13:

| Expression     | Dereferences   | Value (from Figure 6-13)      |
|----------------|----------------|-------------------------------|
| Stack.all      | Entire record  | Top (first) node in the stack |
| Stack.all.Info | The Info field | 'A'                           |
| Stack.all.Next | The Next field | Pointer to second node        |

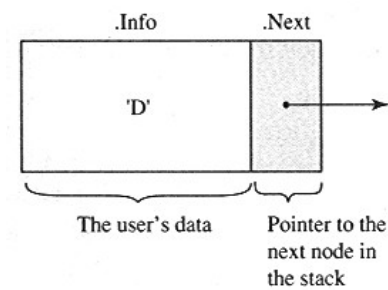


Figure 6-12  
A Single Node

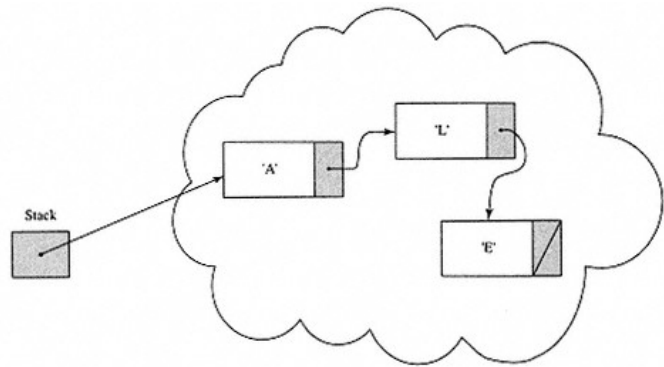


Figure 6-13  
A Stack Composed of Nodes Linked Together

When access types are used to designate records, Ada permits us to leave out the `.all` when dereferencing a field in a designated record. Using this shorthand syntax, the statements above can be rewritten as follows:

| Expression | Dereferences   | Value (from Figure 6-13)      |
|------------|----------------|-------------------------------|
| Stack.all  | Entire record  | Top (first) node in the stack |
| Stack.Info | The Info field | 'A'                           |
| Stack.Next | The Next field | Pointer to second node        |

Because the first statement dereferences the entire record rather than a field, we must include `.all`. This shorthand syntax makes it appear as though `Stack` was a record variable rather than an access type variable. As with the designated array syntax we described in Chapter 5, some Ada programmers prefer the shorthand form because it allows them to easily change their implementation from access variables to record variables and vice versa. Again, we believe that such decisions should be made during program design rather than during program implementation and that the extra documentation provided by `.all` is well worth the extra time it takes to type the four characters. Because of the extra documentation provided, most all of the examples in this text use the `.all` form. You (or your professor) can decide which form is best for your programs. The exercises at the end of this chapter give you practice with both forms.

Let's see how to push a new element onto our linked stack. Figure 6-14 shows the result of pushing 'D' onto the stack illustrated in Figure 6-13. Notice what has changed between these two figures.

- Memory for a new node has been allocated.
- The new element, 'D', has been assigned to the `Info` field of the new node.
- The `Next` field of the new node designates the node previously designated by the `Stack` object.
- The `Stack` object now designates the new node.

All of these changes are carried out by the following Ada statement:

```
Stack := new Node_Type'(Info => New_Element, Next => Stack);
```

Allocates memory for the new node

Assign the new element to Info

Make Next an alias of Stack  
(Next designates top node of the original stack)

Stack designates the new node

This single statement is all that is required to push a new value onto our linked stack. When working with linked data structures such as this stack, it is important to draw pictures such as we have done in Figures 6-13 and 6-14 before attempting to write the Ada statements to carry out an operation on the structure.

### Implementing Procedure Pop

Now let's look at the pop operation. The algorithm for pop is

#### Pop

```
Popped Element := Info in top node
Unlink the top node from the stack
Deallocate the memory used by the old top node
```

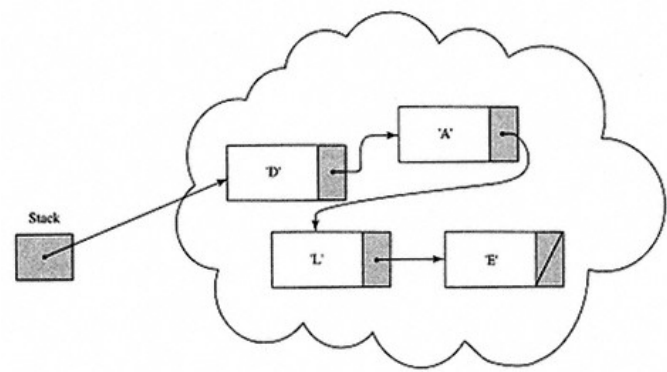


Figure 6-14  
Result of Pushing 'D' onto the Stack of Figure 6-13

Let's try this algorithm with the stack in Figure 6-15. We first put the value from the top node into `Popped_Element`. How do we "unlink" the top node from the stack? If we reset `Stack` to point to the node *following* the top node, the resulting stack should be correct. Now we can deallocate the memory occupied by the old top node by calling a procedure instantiated from `Unchecked_Deallocation`.

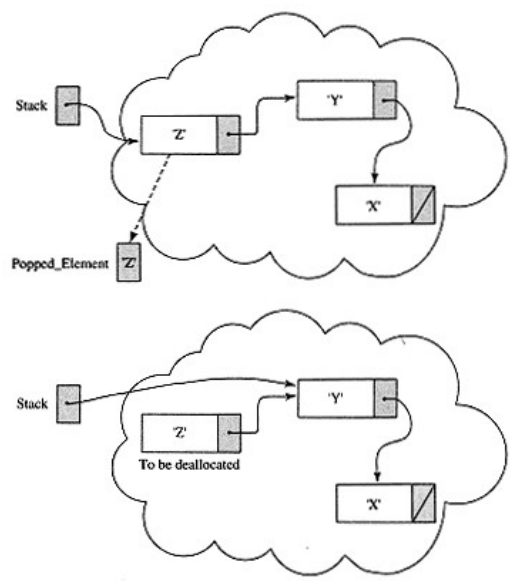


Figure 6-15  
Popping the Top Element

Whoops! The problem with this algorithm is that it leaves the old top node inaccessible—we no longer have an access variable that designates the node. When we code this procedure, let's add a local access variable to designate the old top node before we reset *Stack*. Here's the complete algorithm:

### Pop

```

Popped_Element := Info in top node
Node_To_Recycle := Stack -- Alias of top node
Stack := Next of top element -- Stack designates "2nd" node
Deallocate the memory designated by Node_To_Recycle

```

Let's trace this algorithm, using the stack in Figure 6-16. *Popped\_Element* is set to the value of the *Info* field in the first (top) node [Figure 6-16(a)]. We save an access value to the first node, so that we can access it later for recycling [Figure 6-16(b)]. Then *Stack* is advanced to jump over the first node, making the second node the new top element. How can we get the access value that designates the second node? We get it from the *Next* field of the first node. This access value is assigned to *Stack* to complete the unlinking task [Figure 6-16(c)]. Finally, we deallocate the memory occupied by the old top node by calling a procedure *Free* that was instantiated from *Unchecked\_Deallocation*, using the access value we saved in *Node\_To\_Recycle* [Figure 6-16(d)].

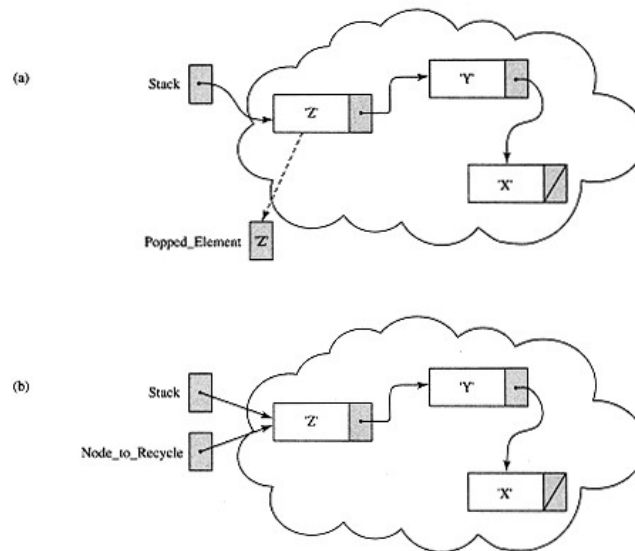
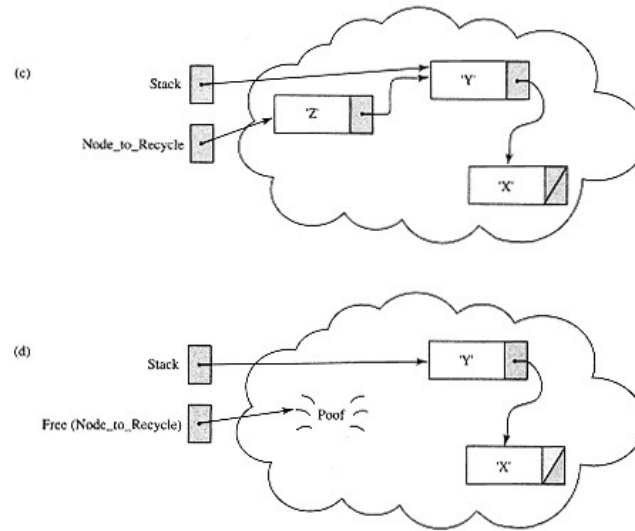


Figure 6-16  
Popping the Stack





Does this algorithm work if there is only one node in the stack when Pop is called? Let's see. `Popped_Element` is assigned the value in the `Info` field in the only node in the stack, and then we unlink this node from the stack. We save an access value to the node, as before, and then try to assign the value of the `Next` field in the first/last node to `Stack` (Figure 6-17). What is the value of this `Next` field? Because this is the last node in the list, its `Next` field contains `null`. This value is assigned to `Stack`, which is exactly what we want because a `null` value for `Stack` means that the stack is empty. So the procedure works for a stack of one element.

### Overflow, Underflow, Empty, and Full

Our algorithms for push and pop did not consider the possibilities of stack overflow or stack underflow. Because we are using Ada's new allocator to obtain memory for a new node, our stack overflows only when the memory available for dynamic allocation is exhausted. The exception `STORAGE_ERROR` is raised when there is no memory available to allocate to the new node. Underflow occurs when we attempt to pop an item off an empty stack. We can use the empty operation to test for underflow in the `Pop` procedure.

The stack is empty when the stack variable does not designate a node. Figure 6-8 shows an empty stack. Function `Empty` simply needs to check if `Stack` has the value `null`. What about function `Full`? It should return `True` if another item *cannot* be pushed onto the stack. This condition can only be detected by trying to allocate new memory and seeing if `STORAGE_ERROR` is raised. If it is raised, the stack is full. Otherwise the stack is not full, and we can deallocate the memory just obtained.

### Implementing Procedure Clear

In the array implementation, this procedure simply set `Top` to zero. In the linked implementation, we could set `Stack` to `null` to leave the stack

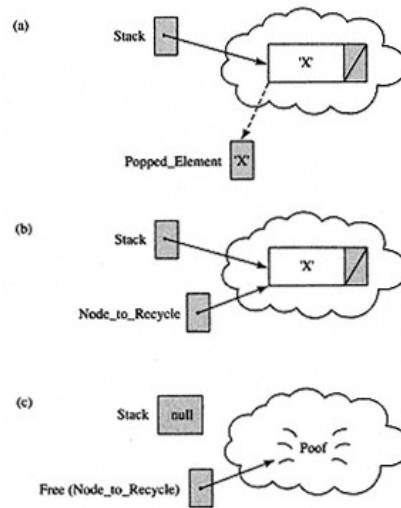


Figure 6-17  
Popping the Last Element in the Stack

empty—but that leaves all the space occupied by the nodes tied up. It's better to free the node space.

Our Clear operation loops through all the elements in the stack, removing them one by one, and disposing of the space.

### Clear

```
loop
 exit when stack is empty
 Unlink the top node
 Free the node
end loop
```

This process is very much like putting procedure Pop inside a loop. In fact, we could just write the loop as

```
loop
 exit when Empty(Stack);
 Pop (Stack, Dummy_Value);
end loop;
```

But for the sake of efficiency (avoiding all the procedure and function calls), not to mention getting some extra practice using access variables, we write the procedure from scratch.

### Linked Stack Package Body

The linked stack implementation requires two declarations: one for the node and another for an access type that designates nodes. Here is how we might declare these types in the private part of our stack package declaration (Specification 6-1):

```

private -- These declarations will not compile

type Stack_Type is access Node_Type; -- Designates a node

type Node_Type is
 record
 Info : Element_Type; -- One item
 Next : Stack_Type; -- Designates the next node in the stack
 end record;

end Stack;

```

A problem here is that the declaration of `Stack_Type` uses `type Node_Type` before `Node_Type` has been declared. Well, let's just swap the order of these two declarations.

```

private -- These declarations will not compile

type Node_Type is
 record
 Info : Element_Type; -- One item
 Next : Stack_Type; -- Designates the next node in the stack
 end record;

type Stack_Type is access Node_Type; -- Designates a node

end Stack;

```

---

#### Incomplete Type Declaration

A type declaration that gives only the name of the type. It may include optional discriminants. An incomplete type may only be used in the declaration of access types.

---

There is still a problem. The access type `Stack_Type` is used in the declaration of `type Node_Type` before `Stack_Type` has been declared. This problem seems impossible to fix. However, we can solve it through the use of an **incomplete type declaration**. Such a declaration contains only the name of the type being declared. Incomplete type declarations may be used only in the declaration of access types. Eventually, every incomplete type declaration must be followed by a full type declaration.

Here are the declarations using an incomplete type declaration for our node:

```

private

type Node_Type; -- Incomplete type declaration

type Stack_Type is access Node_Type;

type Node_Type is -- Full type declaration
 record
 Info : Element_Type; -- One item
 Next : Stack_Type; -- Designates the next node in the stack
 end record;

end Stack;

```

There is an incompatibility with the declaration of `Stack_Type` here and the declaration of `Stack_Type` in the visible part of Specification 6-1 (page 224). There is no discriminant here to correspond with the discriminant in the visible portion. We used the discriminant in the array-based stack package to set the upper bound of the array. We have

no need to set such an upper bound in our linked version (that's its advantage!). We must delete the discriminant in the visible part of the stack specification to use our linked declarations in the private part. Programming problem 2 asks you to write an linked stack body that is completely compatible with Specification 6-1.

Using the declarations above for the private part of our stack package declaration, we implement all the stack operations in the Body 6-2.

### Body 6-2— Linked Stack Implements Specification 6-1

```
with Unchecked_Deallocation;
package body Stack is

 -- Instantiate a procedure to recycle nodes
 procedure Free is new Unchecked_Deallocation (Object => Node_Type;
 Name => Stack_Type);

 procedure Clear (Stack : in out Stack_Type) is

 To_Recycle : Stack_Type; -- Designates a node for recycling

 begin
 loop
 exit when Stack = null; -- Exit when stack is empty
 To_Recycle := Stack; -- Unlink the
 Stack := Stack.all.Next; -- top node
 Free (To_Recycle); -- Recycle the node
 end loop;
 end Clear;

 function Empty (Stack : in Stack_Type) return Boolean is
 begin
 return Stack = null;
 end Empty;

 function Full (Stack : in Stack_Type) return Boolean is

 New_Node : Stack_Type; -- Designates a temporary node

 begin
 New_Node := new Node_Type; -- Get memory for a new node
 Free (New_Node); -- Now recycle the memory
 return False; -- There is still memory available.
 exception
 when STORAGE_ERROR => -- When there is no memory available
 return True; -- the stack is full.
 end Full;
```

```

procedure Push (Stack : in out Stack_Type;
 New_Element : in Element_Type) is
begin
 Stack := new Node_Type' (Info => New_Element,
 Next => Stack);
exception
 when STORAGE_ERROR => -- Stack is full if no memory is available.
 raise OVERFLOW;
end Push;

procedure Pop (Stack : in out Stack_Type;
 Popped_Element : out Element_Type) is

 To_Recycle : Stack_Type; -- Designates a node for recycling

begin
 if not Empty (Stack) then
 Popped_Element := Stack.all.Info; -- Return the top information
 To_Recycle := Stack; -- Unlink the top
 Stack := Stack.all.Next; -- node of the stack
 Free (To_Recycle); -- Recycle the node
 else
 raise UNDERFLOW;
 end if;
end Pop;
end Stack;

```

### ***Comparing the Stack Implementations***

First, let's compare the storage requirements of each of our implementations. An array variable of the maximum stack size takes the same amount of memory, no matter how many array slots actually are used; we need to reserve space for the maximum possible. The linked implementation using dynamically allocated storage space only requires space for the number of elements actually on the stack at run time. Note, however, that the elements are larger, because we must store the link (the `Next` field) as well as the user's data.

Now let's compare the relative program size (number of lines of code) of the two stack implementations. These measurements may give us some idea of how complex each implementation is from the point of view of the programmer. We are assuming that the longer a program is, the more difficult it is to understand. This is not always a valid assumption. A programmer may use some clever trick to reduce many lines of code into a few. Often, the trick is more difficult for someone else to understand than the longer code it replaces. Here are the counts from our two stack implementations:

Comparison of Executable Source Lines of Code

|                             | <i>Array<br/>Implementation</i> | <i>Linked<br/>Implementation</i> |
|-----------------------------|---------------------------------|----------------------------------|
| Declarations (Private Part) | 6                               | 7                                |
| Clear                       | 4                               | 10                               |
| Empty                       | 4                               | 4                                |
| Full                        | 4                               | 10                               |
| Push                        | 10                              | 9                                |
| Pop                         | 10                              | 13                               |
| Total                       | 38                              | 53                               |

As you can see from the table, the array implementation code is shorter than the linked implementation by more than 25 percent.

We can compare the relative efficiency of the two stack implementations in terms of Big-O notation. In both implementations, the `Full` and `Empty` operations are clearly  $O(1)$ . They always take a constant amount of work. What about `Push` and `Pop`? Does the number of elements in the stack affect the amount of work done by these operations? No, it does not. In both implementations, we directly access the top of the stack, so these operations also take a constant amount of work. They too are  $O(1)$ . Only the `Clear` operation differs from one implementation to the other. The array implementation simply sets the `Top` field to zero, so it is clearly an  $O(1)$  operation. The linked implementation must process every node in the stack, in order to recycle the node space. This operation, therefore, is  $O(N)$ , where  $N$  is the number of nodes in the stack. Overall the two stack implementations are roughly equivalent in terms of the amount of work they do, only differing in one of the five operations. Note that if the difference had been in the `Push` or `Pop` operation, rather than the less frequently called `Clear`, it would be more significant. The next table summarizes the Big-O comparison of the stack operations.

Big-O Comparisons of Stack Operations

|       | <i>Array<br/>Implementation</i> | <i>Linked<br/>Implementation</i> |
|-------|---------------------------------|----------------------------------|
| Clear | <b><math>O(1)</math></b>        | <b><math>O(N)</math></b>         |
| Empty | $O(1)$                          | $O(1)$                           |
| Full  | $O(1)$                          | $O(1)$                           |
| Push  | $O(1)$                          | $O(1)$                           |
| Pop   | $O(1)$                          | $O(1)$                           |

Let's look at an actual timed test case, with real numbers. The following table shows how long it takes one particular computer to push and pop 100,000 integer elements. We also pushed 100,000 integer elements on to a stack and timed how long it takes the clear operation to empty the stack.

**Results of the Timed Implementation Comparisons (in seconds)**

|       | <i>Array<br/>Implementation</i> | <i>Linked<br/>Implementation</i> |
|-------|---------------------------------|----------------------------------|
| Push  | 0.36                            | 0.87                             |
| Pop   | 0.32                            | 0.81                             |
| Clear | 0.00*                           | 0.49                             |

The push and pop operations with the array-based stack are faster than the equivalent linked stack operations *on this particular computer*. Do not take these numbers as an absolute measure of the efficiency of these algorithms. The results of timing tests like these are dependent on the compiler used to convert the Ada program to machine language and the computer on which the tests are run. On another computer, the linked versions may run faster. As we predicted, the amount of time to clear an array-based stack is significantly less than the time required to clear a linked stack.

So which implementation is better? The answer, as usual, is, it depends on the situation. The linked implementation certainly gives more flexibility, and in applications where the number of stack elements can vary greatly, it wastes less space when the stack is small. In situations where the stack size is totally unpredictable, the linked implementation is preferable, because size is largely irrelevant. Why then would we ever want to use the array implementation? Because it is short, simple, and efficient. When `Max_Size` is small, and we can be sure that we will not need to exceed the declared stack size, the array implementation is a good choice. Also, if you are programming in a language that does not support dynamic storage allocation, an array implementation may be the only good choice.

**Encapsulation Revisited**

We use private types to encapsulate a class. A private type prevents application programmers from accessing the details of the data structure used to implement the class. They can use only the operations given in the package, equality testing (= and /=), and assignment (: =). A *limited* private type restricts the application programmer even further by limiting operations to those defined in the package; they may not use assignment or equality operators with limited private type objects.

Why is encapsulation desirable? Why not let the application programmer have full access to the details of the data structure used to implement our class? There are two major benefits of encapsulation. First, a great deal of work can be saved later if we decide to change the implementation of our class. Suppose instead of declaring our array-based stack as a private type, we declare it in the public part of our specification. The application programmer is now free to access and modify the record fields and array components. By directly referencing the data structure's details, the application programmer has bound their program to this particular implementation. If at a later time we decide to change our stack to a linked implementation, the application programmer has to change every array reference in their program to pointer references. Such changes might even require major modifications of their design which, as we discussed in Chapter 2, can be an expensive undertaking.

---

\*Too small to measure

A second advantage of encapsulation is reliability. Because the state of an object is modified only through operations defined in the package, we have confidence that our objects are always valid. For example, the state of an array-based stack object is determined by the record field `TOP`. With our encapsulation, only the `Push`, `Pop`, and `Clear` operations can modify this field. We need only apply the lengthy verification and testing methods discussed in Chapter 2 to these three operations to ensure every stack object's validity. The amount of effort to verify *every* direct access of a stack object by an application program that has full access to the data structure makes it less likely that every access is verified.

### ***Private or Limited Private?***

In Specification 6-1 we declared `Stack_Type` as a *limited* private type. This declaration prevents the programmer using our stack class from comparing two stacks with the equality operators (`=` and `/=`) and from using the assignment statement to make copies of stack objects. Why would we want to prevent another programmer from using these operations with stack objects? Although on the surface these operations appear reasonable, when you look at the implementations we developed you find that they yield incorrect results.

First let's look at the equality operators. In both the array-based and linked stacks, comparing two "identical" stack objects with the `=` operator usually returns `False`. The stack elements in our array-based implementation occupy the beginning locations in the array. Unused array elements contain logical garbage (Figure 6-4). Comparing two stack variables with the `=` operator compares the stack elements *and* the logical garbage in each stack. It is very unlikely that the logical garbage in each of two "identical" stacks is the same. Comparing two linked stack objects with the `=` operator compares the two access values that designate the first node in each stack. Only aliases have identical access values; any *two* "identical" stacks have different access values pointing to their first nodes.

There is no problem using the assignment operator with an array-based stack—a copy of the record containing the stack is made. A problem arises when using the assignment operator with the linked stack implementation. As Figure 6-18 shows, using the assignment operator with a linked stack object leaks memory and creates an alias of the original object rather than a copy.

By declaring our stack class as a limited private type, we prevent the application programmer from using the erroneous equality and assignment operators with stack objects. There are some applications that need to create copies of existing stacks and perhaps some that need to see if two stack objects are identical. We can add an `"="` function and a `COPY` procedure to Specification 6-1 to carry out these operations correctly and without memory leaks. We leave the implementation of these subprograms as an exercise.

### **Summary**

We have defined a stack at the logical level as an abstract data type, and we have discussed two implementations of a stack. In the first implementation, an array is used to store the stack elements. In the second implementation, the stack elements are stored in dynamically allocated memory and linked together using access values. By isolating the procedures and functions that operate on the actual representation of the stack in a package and declaring the stack type as a private type, we encapsulate the data structure. No matter which implementation we select, we keep the use of the data structure limited to the interfaces recorded in the stack specification.



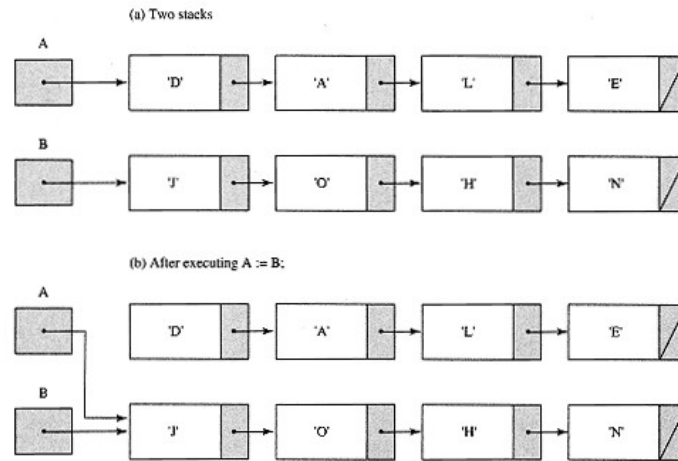


Figure 6-18  
Using the Assignment Operator with Linked Stacks

The independence of implementation from interface resulting from our encapsulation simplifies program maintenance. We don't have to make changes throughout the program whenever we change the stack body.

Though our logical picture of a stack is a linear collection of data elements with the newest element (the top) at one end and the oldest element at the other end, the physical representation of the stack class does not have to recreate our mental image. The implementation of the stack class must support the last in, first out (LIFO) property; how this property is supported, however, is another matter. For instance, the Push operation could time stamp the stack elements, and put them into an array in any order. To pop, we would have to search the array, looking for the newest time stamp. This representation is very different from either of the stack implementations we developed in this chapter, but to the user of the stack class they are all functionally equivalent. The implementation is transparent to the program that uses the stack because the stack is encapsulated by the operations in the package that surround it.

### Exercises

1. Describe the accessing functions of a stack at the abstract level.
2. Show what is written by the following segments of code, given that `Stack` is a stack of integer elements and `X`, `Y`, and `Z` are integer variables:

```

(a) Clear (Stack);
X := 1;
Y := 0;
Z := 4;
Push (Stack, Y);
Push (Stack, X);
Push (Stack, X + Z);
Pop (Stack, Y);
Push (Stack, Z*2);
Push (Stack, Y);
Push (Stack, 3);
Pop (Stack, X);
Integer_IO.Put(X);
Integer_IO.Put(Y);
Integer_IO.Put(Z);
Text_IO.New_Line;
while not Empty(Stack) loop
 Pop(Stack, X);
 Integer_IO.Put(X);
 Text_IO.New_Line;
end loop

(b) Clear (Stack);
X := 4;
Z := 0;
Y := X + 1;
Push (Stack, Y);
Push (Stack, X);
Pop (Stack, Y);
X := Y + 1;
Push (Stack, X);
Push (Stack, Z);
while not Empty (Stack) loop
 Pop (Stack, Z);
 Integer_IO.Put (Z);
 Text_IO.New_Line;
end loop;
Integer_IO.Put (X);
Integer_IO.Put (Y);
Integer_IO.Put (Z);

```

3. You are given the following declarations:

```

subtype Name_String is String (1..20);

type Name_Rec is
 record
 First : Name_String;
 Last : Name_String;
 end record;

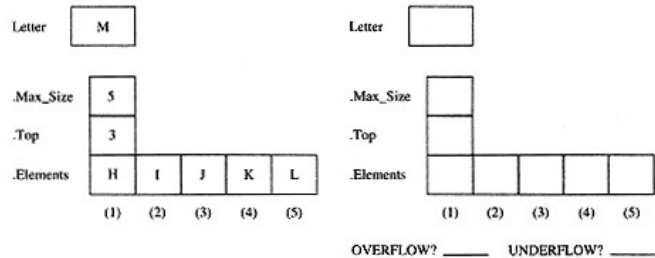
```

- (a) Instantiate a stack package whose elements are Name\_Rec.
- (b) Using the package you just instantiated, declare a stack object that holds a maximum of 250 names.

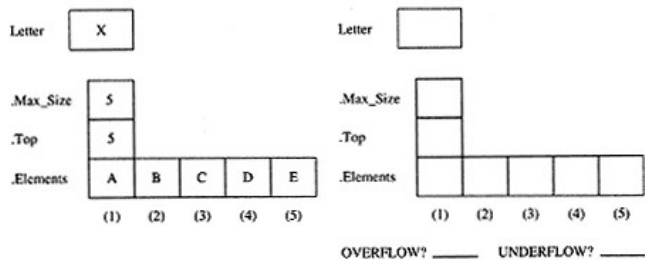
Use the following information for Exercises 4–7:

Stack is implemented as a record containing an array of Elements and a field indicating the index of the Top element, as discussed in this chapter. Letter is a Character variable. In each exercise below, show the result of the operation on the stack. If OVERFLOW or UNDERFLOW is raised, check the appropriate box. In each case show the contents of the array, the Top indicator, Max\_Size, and Letter after the operation is complete. (Note: Some values in the array may not be elements in the stack.)

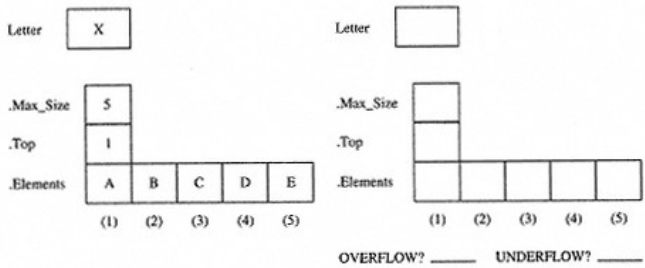
4. Push (Stack, Letter);



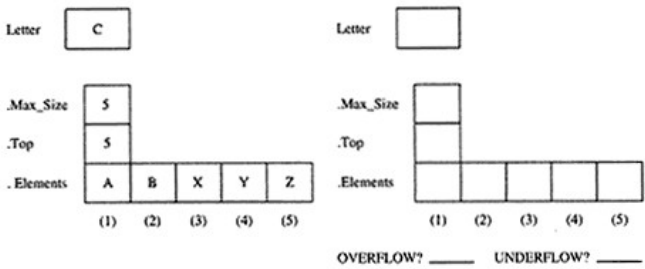
5. Push (Stack, Letter);



6. Pop (Stack, Letter);



7. Pop (Stack, Letter);



8. Write a segment of code to perform each of the following operations. You may call any of the procedures or functions specified for the Stack class. The details of the stack type are encapsulated; you may only use the stack operations in the specification to perform the operations (you may declare additional stack objects).

- (a) Set `Second_Element` to the second element in the stack, leaving the stack without its top two elements.
- (b) Set `Bottom` equal to the bottom element in the stack, leaving the stack empty.
- (c) Set `Bottom` equal to the bottom element in the stack, leaving the stack unchanged.
- (d) Remove all the zero elements from a stack, leaving all the other elements in the stack.

9. *Multiple choice.* If the array implementation of a stack were used, the statement

```
Stack.Elements(1) := Stack.Elements(2);
```

(setting the top element equal to the second element) in a program that uses the stack class

- (a) would cause a syntax error at compile time
- (b) would cause a run-time error
- (c) would not be considered an error by the computer, but would violate the encapsulation of the stack data type
- (d) would be a perfectly legal and appropriate way to accomplish the intended task

10. *Multiple choice.* The statements

```
Push (Stack, X + 1);
Pop (Stack, X + 1)
```

in the main program

- (a) would cause a syntax error at compile time
- (b) would cause a run-time error
- (c) would be legal, but would violate the encapsulation of the stack
- (d) would be perfectly legal and appropriate

11. You are given the following declaration of the `Top` operation:

```
function Top (Stack : in Stack_Type) return Element_Type;

-- Returns the value of the top element without changing the stack.

-- Preconditions : None
--
-- Postconditions : A copy of the top element in the stack is
-- returned.
-- Stack is unchanged.
--
-- Exceptions : UNDERFLOW raised if the stack is empty.
-- The stack is unchanged.
```

Write this function as a new operation in the stack class, using the array-based stack implementation from this chapter.

12. A stack is kept in a record (`Stack_Rec`) that has two fields: `Elements` (an array that contains the stack elements) and `Top` (the index of the current top element). The record variable is not declared, but rather it is allocated dynamically when a stack is needed. Here are the declarations:

```
type Stack_Array is array (Positive range <>) of Element_Type;

type Stack_Rec (Max_Size : Positive) is
 record
 Top : Positive;
 Elements : Stack_Array (1..Max_Size);
 end record;
```

```
type Stack_Type is access Stack_Rec;
```

```
Stack : Stack_Type;
```

- (a) Is the space for this array-based data structure statically or dynamically allocated?
- (b) How much space is reserved for the variable `Stack`?
- (c) Write the `Clear` and `Empty` operations, using the specifications in the `Stack` class.
- (d) Is a `Full` operation for this representation meaningful? If it is, write the `Full` operation. If the operation is not meaningful for this implementation, write a stub.
- (e) If you could write the `Push` and `Pop` procedures by modifying the source code of one of the stack implementations in the chapter, which one would you choose to modify? Write the `Push` and `Pop` operations, using the specifications in the `Stack` class.

13. Two stacks of positive integers are needed, one containing those elements less than or equal to the value 1000 and the other containing those elements that are larger than 1000. The total number of elements in the small-value stack and the large-value stack combined is not more than 200 at any time, but we cannot predict how many occur in each stack. (All the elements could be in the small-value stack, they could be evenly divided, both stacks could be empty, and so on.) Can you think of a strategy to implement both stacks in one array?

- (a) Draw a diagram of how the stack might look.
- (b) Write the declarations for such a double-stack structure.
- (c) Implement the `Push` operation to assign `New_Element` to the correct stack according to its value (compared to 1000).

14. A stack of integer elements is implemented as an array with the following declarations:

```
type Stack_Type is ARRAY (0 .. Max_Stack) of Integer;
```

```
Stack : Stack_Type;
```

The index of the top element is kept in `Stack(0)`, and the stack elements are stored in `Stack(1) .. Stack(Stack(0))`.

- (a) How does this implementation fare with the idea of an array as a homogeneous collection of data elements?
- (b) How would this implementation change the stack specifications? How would it change the procedures and functions?

15. Each node in a linked stack representation points to its successor. What does the last node point to?

Use the following declarations for Exercises 16 and 17:

```
type Node_Type;
type Ptr_Type is access Node_Type;
type Node_Type is
 record
 Info : Integer;
 Next : Ptr_Type;
 end record;
```

```
subtype Stack_Type is Ptr_Type;
```

```
type Info_Array is array (1..10) of Integer;
type List_Data_Type is
 record
 Length : Positive;
 Info : Info_Array;
 end Record;
```

```
type List_Type is access List_Data_Type;
```

```
Stack : Stack_Type;
List : List_Type;
Ptr : Ptr_Type;
```

16. Mark each statement as syntactically valid or invalid.

- (a) Stack.all.Info := 2;
- (b) Stack.Info := 2;
- (c) Stack.Info := List.Info;
- (d) Stack.Info := List.Info(3);
- (e) Ptr := Stack.Next;
- (f) Stack.Info := '2';
- (g) Ptr.Next := Stack;
- (h) List.Length := List.Length + 1;
- (i) Ptr := List;
- (j) Ptr := Ptr.Next;
- (k) List.Info (Ptr.Info) := Stack.Info;

17. Describe each of the following as a pointer, array, record, or integer:

- (a) Stack
- (b) List
- (c) List.Info
- (d) Stack.Info
- (e) Stack.Next
- (f) Ptr.Next
- (g) List.Info (Ptr.Info)

18. Write the body for procedure Print\_Reverse, given the following declaration. Your solution should use a stack. (You may use any of the stack operations defined for the Stack class.)

```
procedure Print_Reverse (Data : in Text_IO.File_Type);

-- This procedure reads a string of characters terminated by a
-- blank from file Data and displays the characters in reverse
-- order on the console.

-- Preconditions : File Data is open for input
--
-- Postconditions : The screen contains characters from file Data,
-- from the original reading marker position up to
-- but not including the first blank, displayed in
-- reverse order. The reading marker for file Data
-- is positioned to read the character following
-- the blank.
```

19. Write a code segment to read in a string of characters and determine whether they form a palindrome. A palindrome is a sequence of characteres that reads the same both forward and backward, for example:

ABLE WAS IERE I SAW ELBA

The character ' . ' ends the string. Write a message indicating whether the string is a palindrome. You may assume that the data is correct and that the maximum number of characters is 80.

20. Write the body for procedure `Replace_Element`, with the following declaration:

```
procedure Replace_Element (Stack : in out Stack_Type;
 Old_El : in Element_Type;
 New_El : in Element_Type);

-- Replace all occurrences of Old_El in the stack with New_El.

-- Preconditions : None
--
-- Postconditions : Stack = original Stack with any occurrence of
-- Old_El changed to New_El
```

(You may use any of the operations defined for the Stack class, but you may not assume any knowledge of how the stack is implemented.)

21. In each plastic container of Pez candy, the colors are stored in random order. Your little brother only likes the yellow ones, so he painstakingly takes out all the candies, one by one, eats the yellow ones, and keeps the others in order so that he can return them to the container in exactly the same order as before—minus the yellow candies, of course. Write the algorithm to simulate this process. (You may use any of the stack operations defined in the Stack class, but may not assume any knowledge of how the stack is implemented.)

22. True or False?

- (a) Because the two Pop implementations developed in the chapter are both  $O(1)$ , they take the same amount of time to execute.
- (b) The array-based stack implementation uses more space than the linked implementation, no matter how many elements are actually in the stack.
- (c) If the elements in a stack are stored in an array, then the stack is necessarily a static variable.

23. Assuming that data of `Element_Type` takes 12 bytes, integers take 2 bytes, and that pointer types take 4 bytes, and `Max_Stack = 100`, compare the space requirements of static array-based versus dynamic linked stack implementations. (In calculating the space requirements of the linked implementation, don't forget to count the external pointer.)

| <i>Number of<br/>Elements</i> | <i>Space for<br/>Static<br/>Array-Based</i> | <i>Space for<br/>Dynamic<br/>Linked Stack</i> |
|-------------------------------|---------------------------------------------|-----------------------------------------------|
| 0                             | _____                                       | _____                                         |
| 10                            | _____                                       | _____                                         |
| 50                            | _____                                       | _____                                         |
| 100                           | _____                                       | _____                                         |

24. Indicate whether a stack would be a logical data structure for each of the following applications.

- (a) A program is to evaluate arithmetic expressions according to the specific order of operators.
- (b) A bank wants to simulate its teller operation to see how waiting times would be affected by adding another teller.
- (c) A program is to receive data that it saves and processes in the reverse order.
- (d) An address book is to be maintained.
- (e) A word processor is to have a PF key that causes the preceding command to be redisplayed. Every time the PF key is pressed, the program is to show the command that preceded the one currently displayed.
- (f) A dictionary of words used by a spelling checker is to be built and maintained.
- (g) A program is to keep track of patients as they check into a medical clinic, assigning patients to doctors on a first-come, first-served basis.
- (h) A data structure is used to keep track of the return addresses for nested procedures while a program is running.

25. Evaluate the following postfix expressions:

(a)  $6\ 4 + 1 - 3 /$

(b)  $1\ 4\ 18\ 6 / 3 ++ 5 / +$

(c)  $1\ 16\ 8\ 4\ 2 /// + 5 *$

26. (a) Modify the postfix expression evaluator (procedure `Postfix`) given in this chapter so that operands may contain more than one digit. The attributes `'Value` and `'Image` can be used to convert between numeric and string types.

(b) Modify the postfix expression evaluator (procedure `Postfix`) given in this chapter so that it uses the stack operation `Empty` to detect invalid postfix expressions. Do *not* count operands and operators.

27. Write a specification for the program `Postfix` given in this chapter.

28. Here is an operation we could add to Specification 6-1 that compares two stacks for equality:

```
function "=" (Left : in Stack_Type;
 Right : in Stack_Type) return Boolean;
```

(a) Write the body for this function using the array-based implementation of Body 6-1.

(b) Write the body for this function using the linked implementation of Body 6-2.

29. Here is an operation we could add to Specification 6-1 that makes one stack object a copy of another:

```
procedure Copy (Target : in out Stack_Type;
 Source : in Stack_Type);
-- Target is a copy of Source
```

(a) Write the body for this function using the array-based implementation of Body 6-1.

(b) Write the body for this function using the linked implementation of Body 6-2. You must first recycle any memory used by stack `Target`. Then you must obtain new memory for each node whose value you copy from stack `Source`. Why is the mode for parameter `Target` `in out` rather than `in`?

30. (a) Why can't we instantiate a package for stacks whose elements are type `Unbounded_String` (Chapter 5)?

(b) What changes can we make to our stack package (specification and body) to accommodate elements like `Unbounded_String`?



## Programming Problems

1. Rewrite program `Postfix` given in this chapter without the input assumptions. You can use `Integer'Value` to convert a string into an integer.

Optional: Allow negative as well as natural operands. The subtraction operator has a blank immediately after it while the minus sign of a negative number has the first digit of the number immediately after it.

2. The following declarations for a linked stack are compatible with the visible part of Specification 6-1:

```
private

type Node_Type;
type Node_Ptr is access Node_Type;
type Node_Type is
 record
 Info : Element_Type;
 Next : Node_Ptr;
 end record;

type Stack_Type (Max_Size : Positive) is
 record
 Top : Node_Ptr; -- Designates the top node in the stack
 end record;

end Stack;
```

(a) Write package body `Stack` using a package specification with these private declarations. The *major* changes from the linked implementation given in the chapter are to functions `Full` and `Empty`.

(b) Write a test plan for unit testing this implementation of the stack class.

(c) Implement your test plan and run it.

3. This problem requires you to write a program to convert an infix expression to postfix format. The evaluation of an infix expression such as  $A + B * C$  requires knowledge of which of the two operations,  $+$  or  $*$ , should be performed first. In general,  $A + B * C$  is to be interpreted as  $A + (B * C)$  unless otherwise specified. We say that multiplication takes *precedence* over addition. Suppose that we would now like to convert  $A + B * C$  to postfix. Applying the rules of precedence, we begin by converting the first portion of the expression that is evaluated, namely the multiplication operation. Doing this conversion in stages, we obtain

$A + B * C$       *Given infix form*

$A + \mathbf{B\ C\ *}$       *Convert the multiplication*

$\mathbf{A\ B\ C\ * +}$       *Convert the addition*

(The part of the expression that has been converted is in bold.)

The major rules to remember during the conversion process are that the operations with highest precedence are converted first and that after a portion of an expression has been converted to postfix it is to be treated as a single operand. Let us now consider the same example with the precedence of operators reversed by the deliberate insertion of parentheses:

( A + B ) \* C      *Given infix form*

**A B + \* C**      *Convert the addition*

A B + C \*      *Convert the multiplication*

Note that in the conversion from "A B + \* C" to "A B + C \*", "A B +" was treated as a single operand. The rules for converting from infix to postfix are simple, provided that you know the order of precedence.

We consider four binary operations: addition, subtraction, multiplication, and division. These operations are denoted by the usual operators, +, -, \*, and /, respectively. There are two levels of operator precedence. Both \* and / have higher precedence than + and -. Furthermore, when unparenthesized operators of the same precedence are scanned, the order is assumed to be left to right. Parentheses may be used in infix expressions to override the default precedence.

As we discussed in this chapter, the postfix form requires no parentheses. The order of the operators in the postfix expressions determines the actual order of operations in evaluating the expression, making the use of parentheses unnecessary.

### **Input**

The input file contains a collection of *error-free* infix arithmetic expressions, one expression per line. Expressions are terminated by semicolons, and the final expression is followed by a period. An arbitrary number of blanks may occur between any two symbols in an expression. A symbol may be an operand (a single uppercase letter), an operator (+, -, \*, or /), a left parenthesis, or a right parenthesis.

### **Sample Input**

```
A + B - C ;
A + B * C ;
(A + B) / (C - D) ;
((A + B) * (C - D) + E) / (F + G) .
```

### **Output**

Your output should consist of each input expression, followed by its corresponding postfix expression. All output (including the original infix expressions) must be clearly formatted (or reformatted) and also clearly labeled.

### **Sample Output**

```
Infix: A + B - C ;
Postfix: A B + C -
```

```
Infix: A + B * C ;
Postfix: A B C * +
```

```
Infix: (A + B) / (C - D) ;
Postfix: A B + C D - /
```

```
Infix: ((A + B) * (C - D) + E) / (F + G) .
Postfix: A B + C D - * E + F G + /
```

### **Discussion**

In converting infix expressions to postfix notation, the following fact should be taken into consideration: In infix form the order of applying operators is governed by the possible appearance

of parentheses and the operator precedence relations; however, in postfix form the order is simply the "natural" order—the order of appearance from left to right.

Accordingly, subexpressions within innermost parentheses must first be converted to postfix, so that they can then be treated as single operands. In this fashion, parentheses can be successively eliminated until the entire expression has been converted. The *last* pair of parentheses to be opened within a group of nested parentheses encloses the *first* subexpression within that group to be transformed. This last-in, first-out behavior should immediately suggest the use of a stack. Your program may utilize any of the operations in the Stack class.

In addition, you must devise a Boolean function that takes two operators and tells you which has higher precedence. This function is helpful because in rule 3 below you need to compare the next input symbol to the top stack element. Question: What precedence do you assign to '('? You need to answer this question since '(' may be the value of the top element in the stack.

You should formulate the conversion algorithm using the following six rules:

*Rule 1:* Scan the input string (infix notation) from left to right. One pass is sufficient.

*Rule 2:* If the next symbol scanned is an operand, it may be immediately appended to the postfix string.

*Rule 3:* If the next symbol is an operator, then do the following:

- (a) Pop and append to the postfix string every operator on the stack that
  - (i) is above the most recently scanned left parenthesis, and
  - (ii) has precedence higher than or equal to that of the new operator symbol.
- (b) Then push the new operator symbol onto the stack.

*Rule 4:* When an opening (left) parenthesis is seen, it must be pushed onto the stack.

*Rule 5:* When a closing (right) parenthesis is seen, all operators down to the most recently scanned left parenthesis must be popped and appended to the postfix string. Furthermore, this pair of parentheses must be discarded.

*Rule 6:* When the infix string is completely scanned, the stack may still contain some operators. (No parentheses at this point. Why?) All these remaining operators should be popped and appended to the postfix string.

## Classes

An object analysis of the preceding specification and discussion suggests that our problem contains the following *objects*:

Operators

Operands

Infix expressions

Postfix expressions

Input lines

Output lines

Subexpressions

Stack

What predefined or programmer-defined *classes* can we use as templates for these objects? Operators and operands are both characters. The stack object is obviously an instance of the stack class developed in this chapter. The remaining objects are strings.

You may use either stack implementation from the chapter and may extend it with additional operations if necessary—just be sure that they are not specific to this particular application (for example, `procedure Rule_2`) and that you document their specifications well. You may use Ada's predefined type `String` or either of the string classes from Chapter 5.

### Examples

Here are two examples to help you understand how the algorithm works. Each line on the following page demonstrates the state of the postfix string and the stack when the corresponding next infix symbol is scanned. The rightmost symbol of the stack is the top symbol. The rule number corresponding to each line demonstrates which of the six rules was used to reach the current state from that of the previous line.

*Example 1:* Input expression is  $A + B * C / D - E$ .

| Next symbol | Postfix string    | Stack | Rule |
|-------------|-------------------|-------|------|
| A           | A                 |       | 2    |
| +           | A                 | +     | 3    |
| B           | A B               | +     | 2    |
| *           | A B               | + *   | 3    |
| C           | A B C             | + *   | 2    |
| /           | A B C *           | + /   | 3    |
| D           | A B C * D         | + /   | 2    |
| -           | A B C * D / +     | -     | 3    |
| E           | A B C * D / + E   | -     | 2    |
|             | A B C * D / + E - |       | 6    |

*Example 2:* Input expression is  $(A + B * (C - D)) / E$ .

| Next symbol | Postfix string    | Stack     | Rule |
|-------------|-------------------|-----------|------|
| (           |                   | (         | 4    |
| A           | A                 | (         | 2    |
| +           | A                 | ( +       | 3    |
| B           | A B               | ( +       | 2    |
| *           | A B               | ( + *     | 3    |
| (           | A B C             | ( + * (   | 4    |
| C           | A B C             | ( + * (   | 2    |
| -           | A B C D           | ( + * ( - | 3    |
| D           | A B C D -         | ( + * ( - | 2    |
| )           | A B C D - * +     | ( + *     | 4    |
| )           | A B C D - * +     |           | 4    |
| /           | A B C D - * + E   | /         | 3    |
| E           | A B C D - * + E   | /         | 2    |
|             | A B C D - * + E / |           | 6    |

### Turn In

1. program listing
2. output of test run of the program

7—

**FIFO Queues****GOALS**

After reading this chapter, you should be able to

- describe the structure of a FIFO queue and its operations at the abstract level
- determine when a FIFO queue is an appropriate data structure for a specific problem
- design and implement the solution to a problem for which a FIFO queue is an appropriate data structure
- use a static array to implement a FIFO queue class
- use a linked structure to implement a FIFO queue class
- compare FIFO implementations in terms of source code length, use of storage space, and Big-O approximations of the queue operations
- choose between a static and a dynamic linked FIFO queue structure

In Chapter 6 we talked about a stack, a collection class with the special property that elements are always added to and removed from the top. We know from experience that many collections of data elements operate in the reverse manner: Elements are added at one end and removed from the other. This structure, called a **FIFO (First In, First Out) queue**, has many uses in computer programs. In this chapter we consider the FIFO queue data structure at three levels: abstract, application, and implementation. In the rest of this chapter, "queue" refers to a FIFO queue. (Another queue-type data structure, the priority queue, is discussed in Chapter 13. The accessing function of a priority queue is different from that of a FIFO queue.)

## The Abstract Level

### What Is a Queue?

#### FIFO Queue

A data structure in which elements are added to the rear and removed from the front; a "first in, first out" (FIFO) structure.

What is a **queue**? A queue (pronounced like the letter Q) is an ordered, homogeneous group of elements in which new elements are added at one end (the "rear") and elements are removed from the other end (the "front"). As an example of a queue, consider a line of students waiting to pay for their textbooks at a university bookstore (see Figure 7-1). In theory, if not in practice, each new student gets in line at the rear. When the cashier is ready for a new customer, the student at the front of the line is served.

To add elements to a FIFO queue we access the rear of the queue; to remove elements we access the front. The middle elements are logically inaccessible. It is convenient to picture the queue as a linear structure with the front at one end and the rear at the other end. However, we must stress that the "ends" of the queue are abstractions; they may or may not correspond to any physical characteristics of the class's implementation. The essential property of the queue is its FIFO access.

Like the stack, the queue is a holding structure for data that we will use later. We put a data element onto the queue, and then when we need it we remove it from the queue. If we want to change the value of an element, we must take that element off the queue, change its value, and then return it to the queue. We do not directly manipulate the values of elements that are currently in the queue.



Figure 7-1  
A FIFO Queue

### Operations on FIFO Queues

The bookstore example suggests two operations that can be applied to a queue. First, we can add new elements to the rear of the queue, an operation that we call *Enqueue*. We also can take elements off the front of the queue, an operation that we call *Dequeue*. Unlike the stack operations Push and Pop, the adding and removing operations on a queue do not have standard names. Enqueue is sometimes called Enq, Add, or Insert; Dequeue also is called Deq, Remove, or Serve.

Another useful queue operation is checking whether the queue is empty. The *Empty* function returns True if Queue is empty and False otherwise. We can only Dequeue when the queue is not empty. Theoretically we can always Enqueue, for in principle a queue is not limited in size. We know from our experience with stacks, however, that certain

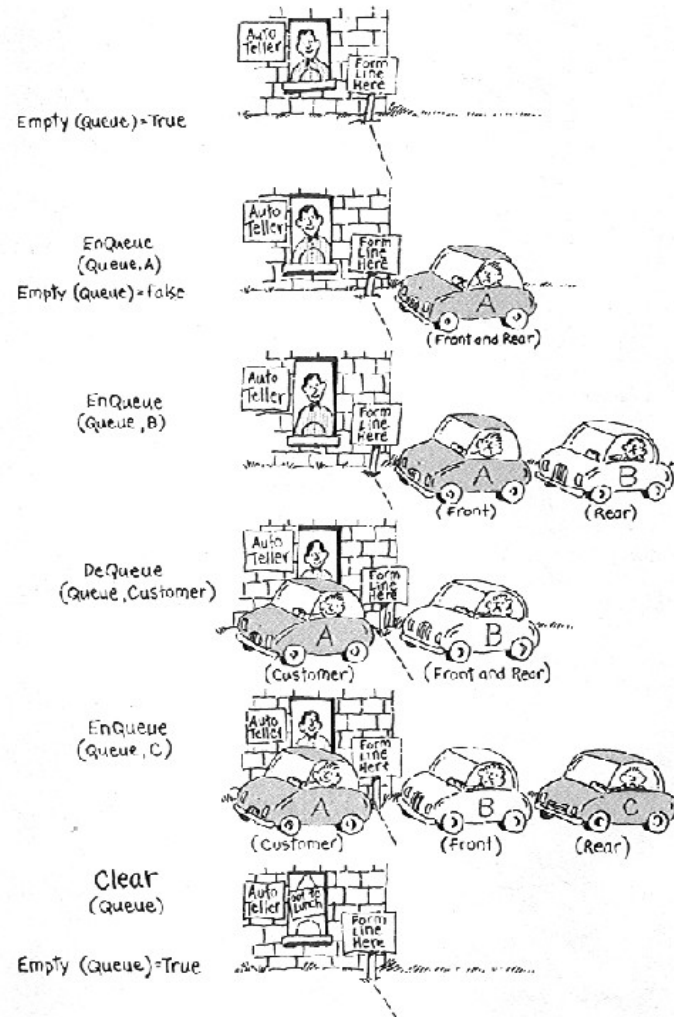


Figure 7-2  
The Effects of Queue Operations

implementations (an array representation, for instance) have an upper limit on the number of elements that a particular collection class object can hold. This real-world consideration applies to queues as well, so we define a *Full* operation. As with stacks, we also might want an operation to destroy the whole structure, leaving it empty. We call this operation *Clear*. Figure 7-2 shows how a series of these operations would affect a queue. We briefly have described a set of accessing operations for a FIFO queue.

Next we must formalize the queue operations that we have described. As with the stack class, we'll make this a generic package so that our queue abstraction can be easily reused in different applications. We can instantiate queue packages for queues containing any type of objects. The declaration of our queue ADT is given in Specification 7-1.

#### Specification 7-1— FIFO Queue Class

```
generic

 type Element_Type is private;

package Queue is

 -- This package implements FIFO queue, a data structure in which
 -- elements are added to the rear and removed from the front;
 -- a "first in, first out" (FIFO) structure.

 type Queue_Type (Max_Size : Positive) is limited private; -- The queue class

 OVERFLOW : exception;
 UNDERFLOW : exception;

 procedure Clear (Queue : in out Queue_Type);
 -- This procedure sets the queue to empty.

 -- Preconditions : None
 --
 -- Postconditions : Queue is empty

 procedure Enqueue (Queue : in out Queue_Type;
 Item : in Element_Type);
 -- This procedure adds Item to the queue.

 -- Preconditions : None
 --
 -- Postconditions : Queue = original Queue with Item added to rear
 --
 -- Exceptions : OVERFLOW is raised if there is no room in the queue
 -- for the Item. The queue is not changed.

 procedure Dequeue (Queue : in out Queue_Type;
 Item : out Element_Type);
```



```

-- This procedure removes and returns the first element in the queue.

-- Preconditions : None
--
-- Postconditions : Queue = original Queue with front element removed
-- Item = front element of original queue
--
-- Exceptions : UNDERFLOW is raised if the queue is empty.
-- The queue is not changed.

function Full (Queue : in Queue_Type) return Boolean;
-- This function determines whether the queue is full or not.

-- Preconditions : None
--
-- Postconditions : Full = (Queue is full)

function Empty (Queue : in Queue_Type) return Boolean;
-- This function determines whether the queue is empty or not.

-- Preconditions : None
--
-- Postconditions : Empty = (Queue is empty)

private

-- We'll fill this in later.

end Queue;

```

---

**Queue Underflow**

The condition resulting from trying to dequeue an empty queue.

**Queue Overflow**

the condition resulting from trying to enqueue an element onto a full queue.

---

Like our stack package in Chapter 6, this package contains the declarations for two exceptions. If a queue is empty when we try to dequeue an element from it, the resulting error condition is called **queue underflow**. The exception comments for procedure `Dequeue` indicate that the exception `UNDERFLOW` is raised if this condition is detected. **Queue overflow** is the condition resulting from trying to enqueue a new element onto a queue that is already full. Procedure `Enqueue` raises the exception `OVERFLOW` if this condition is detected. Our exception comments specify exactly what the state of the queue is after the package raises the exception.

### The Application Level

FIFO queues are commonly used as "waiting lines." Such waiting lines are common on multiuser computer systems and networked systems of workstations. If you are using a multiuser or networked computer system, you probably share a printer with other users. When you request a print out of a file, your request is added to a print queue. When your request gets to the front of the print queue, your file is printed. The print queue ensures that

only one person at a time has access to the printer and that they get this access on a first come, first served basis. Similarly, queues are used to schedule use of other shared resources like disks and tapes.

Another application area in which queues figure as the prominent data structure is the computer simulation of real-world situations. For instance, consider a bank that is planning to set up drive-in teller windows. There should be enough tellers to service each car within a reasonable wait time, but not too many tellers for the number of customers. The bank may want to run a computer simulation of typical customer transactions, using objects to represent the real-world physical objects like tellers, cars, and the clock. The FIFO queue is used to represent the waiting line of customers.

### ***Freight Train Manifests***

The DCT railroad provides service for Canadian companies shipping goods into the United States. At Rouses Point, New York, the railroad operates a facility called a hump yard for organizing incoming cars into outgoing trains. In such a rail yard, cars are released on the top of a hill (the hump). As they roll down the hill, they are switched into the section of track in which cars going to the same destination are collected.

Figure 7-3 illustrates a portion of the Rouses Point hump yard. Cars destined for Austin are collected on one track, for Bethesda on the second track, and for Plattsburgh on the third track. Each of these track sections can hold up to 50 cars. In this yard, a locomotive takes cars one at a time from the incoming car track to the top of the hill and releases them. As they roll down the hill, the cars are switched into the proper section of track. After the locomotive has transferred all of the incoming cars to the yard a locomotive is coupled to each group of cars going to the same destination. Each engineer of the three outgoing locomotives is given a list of cars in their train (a manifest). Our problem is to print a manifest for each of the engineers taking trains out of the yard. See the top of page 273 for a more complete specification.

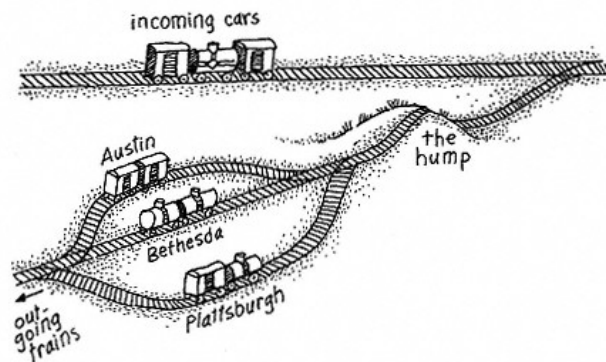


Figure 7-3  
The Rouses Point Hump Yard

## Specification: Freight Train Manifests

### Function

Print a manifest (list of cars) for trains leaving the Rouses Point hump yard. The program should terminate when there are no more cars remaining on the incoming car track. An outgoing train is complete when its track section is filled or there are no more cars remaining on the incoming car track.

### Input

A text file, Cars.Dat, containing a car identification number, car class (box, flat, hopper, tank), and destination (Austin, Bethesda, or Plattsburgh) of all cars on the incoming car track. There is one line in the file for each car.

### Output

The destination and a list of car identification numbers and car classes for each outgoing train. The cars should be listed in order starting at the locomotive.

### Assumptions:

1. Each track section in the hump yard can hold up to 50 cars.
2. Car classes and destinations given in the data file are all valid.

### Object Classes

The obvious object classes in this problem are the yard, cars, locomotives, trains, track sections, and manifests. A car can be modeled as a record with three fields: ID, car type, and destination. Here are the necessary type declarations.

```
type Car_Class_Type is (Box, Flat, Hopper, Tank);
type Destination_Type is (Austin, Bethesda, Plattsburgh);
subtype Car_ID_Type is Positive;

type Car_Rec is
 record
 ID : Car_ID_Type;
 Class : Car_Class_Type;
 Destination : Destination_Type;
 end record;
```

Track sections contain groups of cars. Because the data file contains all the information for the cars on the incoming car track, we use `Text_IO.File_Type` to model the incoming car track. How about the sections of track in the hump yard? We could use an array of car records to model a hump yard track section. However, because the elements in an array can be accessed in any order, the array is not an accurate model of a section of track. We can add cars to a track section only from one end (the end nearest the hump) and remove them only from the other end. A track section is like a waiting line, the first car in is the first car out. A queue is therefore a more accurate model of a track section. Here are the necessary declarations:

```

package Car_Queue is new Queue (Element_Type => Car_Rec);

Section_Size : constant := 50; -- Maximum cars per section
subtype Track_Section is Car_Queue.Queue_Type (Max_Size => Section_Size);

```

The yard itself is a collection of track sections. Because we need to access the various sections randomly, an array of track sections is an appropriate model. There is one track section in the yard for each of the destinations. We can use the destination as an index for the array. Here are the declarations for the yard class and a single yard object.

```

type Yard_Array is array (Destination_Type) of Track_Section;

```

```

Yard : Yard_Array;

```

The manifests are the output of the program. We can use the standard output file for the manifests. Locomotives and trains were mentioned in the problem statement. However, as you shall see in our algorithm design, they are not necessary for our model of the train yard. Eliminating objects that are not essential in a model is an important aspect of object-oriented design.

### Algorithm Design

Now we are ready to design the algorithm for the program (the director object). We base the algorithm for the program on the actual operation of the yard. Cars are taken one at a time from the incoming car track and sent onto the appropriate track section in the hump yard. If a track section is filled an outgoing train is ready to depart, so a manifest is printed and the track section is emptied. When the incoming track is empty, all cars in the yard are sent out. Here is the algorithm in top-down form:

#### Hump Yard

*Level 0*

```

Open Input File
loop
 exit when Input File is empty
 Get Car from Input File
 Add Car to Appropriate Track Section
 if the Track Section is full
 Print Manifest
 end if
end loop
Close Input File
for each non-empty Track Section
 Print Manifest
end loop

```

#### Add Car to Appropriate Track Section

*Level 1*

```

Enqueue (Queue => Yard (Destination),
 Item => Car)

```

**Print Manifest (Destination, Track Section)***Level 1*

```

Put Destination
loop
 exit when Track Section is empty
 Dequeue Car from Track Section
 Display Car information
end loop

```

Note how the destination of a car is used as an array index to select the appropriate track section (queue) to add (enqueue) the car. Program 7-1 implements this algorithm.

**Program 7-1—  
Rouses Point Hump Yard**

```

with Queue;
with Integer_IO;
with Text_IO;
procedure Hump_Yard is

-- This program prints manifests (lists of cars) for trains
-- leaving the Rouses Point hump yard.

type Car_Class_Type is (Box, Flat, Hopper, Tank);
type Destination_Type is (Austin, Bethesda, Plattsburgh);
subtype Car_ID_Type is Positive;

package Class_IO is new Text_IO Enumeration_IO (Enum => Car_Class_Type);
package City_IO is new Text_IO Enumeration_IO (Enum => Destination_Type);

-- Type for railroad cars
type Car_Rec is
 record
 ID : Car_ID_Type;
 Class : Car_Class_Type;
 Destination : Destination_Type;
 end record;

-- Types for track sections in the rail yard

package Car_Queue is new Queue (Element_Type => Car_Rec);

Section_Size : constant := 50; -- Maximum cars per section
subtype Track_Section is Car_Queue.Queue_Type (Max_Size => Section_Size);

-- Type for the yard (a collection of track sections)
type Yard_Array is array (Destination_Type) of Track_Section;

Yard : Yard_Array; -- The hump yard
Car : Car_Rec; -- A single railroad car
Incoming : Text_IO.File_Type; -- Cars on the incoming car track

```

```

procedure Print_Manifest (Destination : in Destination_Type;
 Track : in out Track_Section) is
-- This procedure displays information on the cars on the given track.

-- Preconditions : None
--
-- Postconditions : The information for the cars on the given track is displayed.
-- Track is empty.

 Car : Car_Rec; -- Information on one car

begin
 -- Display headings
 Text_IO.Put ("Manifest for train going to ");
 City_IO.Put (Destination);
 Text_IO.New_Line(2);
 Text_IO.Put ("Car ID");
 Text_IO.Set_Col (To => 10);
 Text_IO.Put ("Car Class");
 Text_IO.New_Line(2);

 loop -- Each iteration, the information for one car is displayed
 exit when Car_Queue.Empty (Track);
 -- Assertion: The queue Track is not empty
 Car_Queue.Dequeue (Queue => Track, Item => Car);
 Integer_IO.Put (Item => Car.ID, Width => 5);
 Text_IO.Set_Col (To => 10);
 Class_IO.Put (Car.Class);
 Text_IO.New_Line;
 end loop;
 Text_IO.New_Page;
end Print_Manifest;

begin -- Hump Yard
 Text_IO.Open (File => Incoming,
 Mode => Text_IO.In_File,
 Name => "Cars.Dat");

 -- Assertion: No queue is full.

 Input_Loop: -- Each iteration, one car is moved to the
loop -- appropriate track section in the yard.
 exit Input_Loop when Text_IO.End_Of_File (Incoming);
 -- Get the information for a car
 Integer_IO.Get (File => Incoming, Item => Car.ID);
 Class_IO.Get (File => Incoming, Item => Car.Class);
 City_IO.Get (File => Incoming, Item => Car.Destination);
 Text_IO.Skip_Line (Incoming); -- Advance reading marker
 -- Add the car to the appropriate track section
 Car_Queue.Enqueue (Queue => Yard (Car.Destination),
 Item => Car);

```

```

-- Check for a full track section
if Car_Queue.Full (Yard (Car.Destination)) then
 Print_Manifest (Destination => Car.Destination,
 Track => Yard (Car.Destination));
end if;

-- Assertion: No queue is full.

end loop Input_Loop;
Text_IO.Close (Incoming);

-- Display manifests for all trains (from non-empty track sections)
for Destination in Destination_Type loop
 if not Car_Queue.Empty (Yard (Destination)) then
 Print_Manifest (Destination => Destination,
 Track => Yard (Destination));
 end if;
end loop;
end Hump_Yard;

```

This program does not have any exception handlers for queue `OVERFLOW` or `UNDERFLOW`. *There is no chance of these exceptions being raised.* How can we write this statement with such confidence? Let's look at the assertion comments we included in this program. The first assertion in the main program is true because the queues were created empty, they have a maximum size greater than zero, and we have not enqueued any values into any queue at this point. Therefore there is no possibility that the first call to procedure `Enqueue` can raise `OVERFLOW`. The second assertion in the main program is always true because if the call to `Enqueue` a few lines earlier filled a queue, the call to `Print_Manifest` empties it. Therefore there is no possibility that the next call to `Enqueue` can raise `OVERFLOW`. Only procedure `Dequeue` raises `UNDERFLOW`. This operation is only called in `Print_Manifest`. Do you see why the assertion made in this procedure is true? This assertion guarantees that `UNDERFLOW` is never raised in this program.

The assertions in this program are based on the results of explicit tests of a queue's state through the `Full` and `Empty` operations. Why did we explicitly test for full and empty queues in this application instead of writing exception handlers as we did in the postfix expression evaluation application in Chapter 6? In this problem, full and empty queues are expected to occur; testing for these states is part of the algorithm. The postfix expression application of Chapter 6 expects the stack to be empty at only one point—the end of the evaluation where we test it by calling `Empty`. A full or empty stack at other times is not expected; such an occurrence is considered an exceptional event. We reserve exception handlers for such unexpected events and use explicit tests when it is part of the processing algorithm.

Now you might raise the question of why we bothered to include the exceptions `OVERFLOW` and `UNDERFLOW` in our FIFO queue package specification when our application does not use them. When we design a class, we try to make it as general as possible so that we can reuse it in many different applications. In the stack and FIFO queue classes, we have supplied both exceptions and explicit operations that return the state of the class so that the application designer can choose the method most appropriate for the problem.

### The Implementation Level

Now that we've had the opportunity to be queue users, let's take a look at how the body of the FIFO queue might be implemented. As with a stack, the queue can be stored in a static variable, an array, as well as in dynamically allocated memory in a linked structure. We look at both implementations in this chapter. Note that the application programmer using our queue should not have to be concerned about which implementation is used, because the interfaces to all the queue operations is the same.

#### The Implementation of a Queue As a Static Array

Like a stack, a queue can be stored in a record with the elements in an array and other information in separate fields.

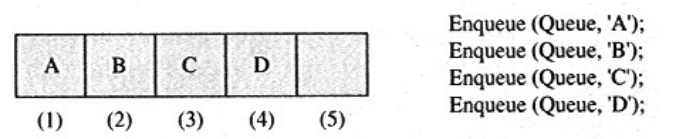
```

type Queue_Array is array (Positive range <>) of Element_Type;

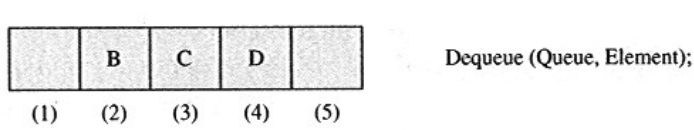
type Queue_Type (Max_Size : Positive) is
 record
 Items : Queue_Array (1..Max_Size); -- The element array
 . -- Other information goes here
 end record;
```

The first question to consider is how we order the elements in the array. In implementing the stack in Chapter 6, we began by inserting an element in the first array position and then let the top float with subsequent Push and Pop operations. The bottom of the stack, however, was fixed at the first slot in the array. Can we use a similar solution for a queue, keeping the front of the queue fixed in the first array slot and letting the rear move down as we add new elements?

Let's see what happens after a few Enqueues and Dequeues if we insert the first element in the first array position, the second element in the second position, and so on. After four calls to Enqueue, the queue would look like this:

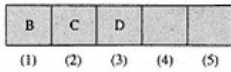


Remember that the front of the queue is fixed at the first slot in the array, whereas the rear of the queue moves down with each Enqueue. Now we Dequeue the front element in the queue:



This operation deletes the element in the first array slot and leaves a hole. To keep the front of the queue fixed at the top of the array, we need to move every element in the queue up one slot:





Let's summarize the queue operations corresponding to this queue design. The Enqueue operation is the same as Push. The Dequeue operation is more complicated than Pop, because all the remaining elements of the queue have to be shifted up in the array, to move the new front of the queue up to the first array slot. However, the Clear, Empty, and Full operations could be the same as the equivalent stack operations.

Before we go any further, we want to stress that this design would work. It may not be the best design for a queue, but it could be successfully implemented. There are multiple *functionally correct* ways to implement the same abstract data structure. One design may not be as good as another (because it uses more space in memory or takes longer to execute) and yet still may be correct. Though we don't advocate the use of poor designs for programs or data structures, the first requirement must always be program correctness.

Now let's evaluate this design. Its strength is its simplicity; it is almost exactly like the package of stack routines that we wrote in Chapter 6. Though the queue is accessed from both ends rather than just one (as in the stack), we only have to keep track of the rear, because the front is fixed. Only the Dequeue operation is more complicated. What is the weakness of the design? The need to move all the elements up every time we remove an element from the queue increases the amount of work needed to Dequeue.

How serious is this weakness? To make this judgment, we have to know something about how the queue will be used. If this queue will be used for storing large numbers of elements at one time, or if the elements in the queue will be large (records with many fields, for instance), the processing required to move up all the elements after the front element has been removed makes this solution a poor one. On the other hand, if the queue generally contains only a few elements and they are small (integers, for instance), all this data movement may not amount to much processing. Further, we need to consider whether performance—how fast the program executes—is of importance in the application that will use the queue. Thus the complete evaluation of the design depends on the requirements of the program.

In the real programming world, however, you don't always know the exact uses or complete requirements of programs. For instance, you may be working on a very large project with a hundred other programmers. Other programmers may be writing the specific application programs for the project while you are producing some packages that will be used by all the different applications. If you don't know the requirements of the various users of your queue package, you must design it for the most general case. In this situation the design just described is not the best one possible.

### ***Another Array Design***

The need to move the elements in the array was created by our decision to keep the front of the queue fixed in the first array slot. If we keep track of the index of the front as well as the rear, we can let both ends of the queue float in the array. Here are the algorithms of Enqueue and Dequeue for this design:

#### **Enqueue**

```
Increment Rear
Add the new element at the array location indicated by Rear
```

## Dequeue

Take the new element at the array location indicated by Front  
Increment Front

Figure 7-4 shows how several Enqueues and Dequeues would affect the queue. (For simplicity, these figures show only the elements that are in the queue. The other slots contain logical garbage, including values that have been Dequeued. The Enqueue operations have the same effect as before; they add elements to subsequent slots in the array and increment the index of the Rear indicator. The Dequeue operation is simpler, however. Instead of moving elements up to the beginning of the array, it merely increments the Front indicator to the next slot.

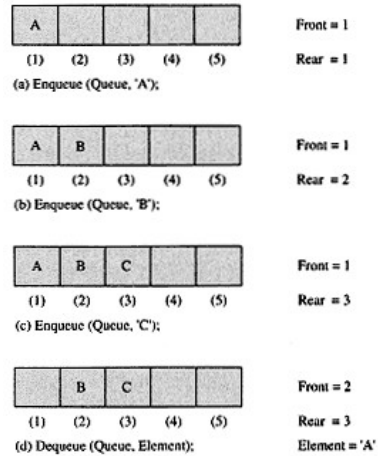


Figure 7-4  
The Effect of Enqueue and Dequeue

Letting the queue elements float in the array creates a new problem when Rear gets to the end of the array. In our first design this situation told us that the queue was full. Now, however, it is possible for the rear of the queue to reach the end of the (physical) array when the (logical) queue is not yet full [Figure 7-5(a)].

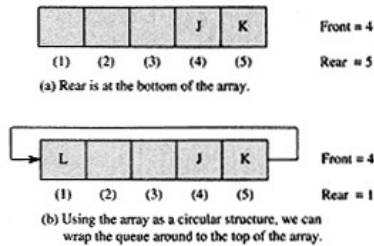


Figure 7-5  
Wrapping the Queue Around

Because there may still be space available at the beginning of the array, the obvious solution is to let the queue "wrap around" the end of the array. In other words, the array can be treated as a circular structure, in which the last slot is followed by the first slot [Figure 7-5(b)]. To get the next position for `Rear`, for instance, we can use an if statement.

```
if Rear = Max_Size then
 Rear := 1;
else
 Rear := Rear + 1;
end if;
```

Using the `rem` operator, the logic expressed in this if statement can be done with the following single assignment:

```
Rear := Rear rem Max_Size + 1;
```

Adding array wrap-around gives us the following revised algorithms:

### Enqueue

```
Rear := (Rear rem Max_Size) + 1
Add the new element at the array location indicated by Rear
```

### Dequeue

```
Take the new element at the array location indicated by Front
Front := (Front rem Max_Size) + 1
```

This solution leads us to a new problem: How do we know whether a queue is empty or full? In Figure 7-6 we remove the last element, leaving the queue empty. In Figure 7-7 we add an element to the last free slot in the queue, leaving the queue full. The values of `Front` and `Rear`, however, are identical in the two situations.

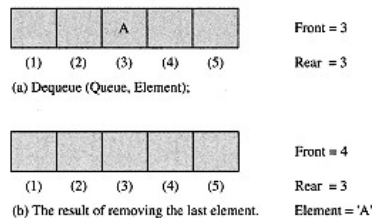


Figure 7-6  
An Empty Queue

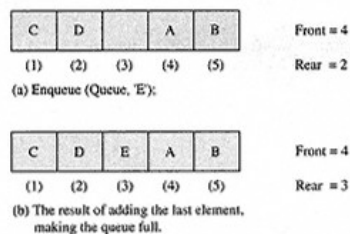


Figure 7-7  
A Full Queue

A simple solution that comes to mind is to add another field to our queue record, in addition to `Front` and `Rear`—a count of the elements in the queue. When the count field is zero, the queue is empty; when the count is equal to the maximum number of array slots, the queue is full. Of course we must remember to increment `Count` in the `Enqueue` operation and decrement it in the `Dequeue` operation. Here are the declarations for the private part of Specification 7-1 (page 270) using our solution of array wrap-around and a count field.

```
private

type Queue_Array is array (Positive range <>) of Element_Type;

type Queue_Type (Max_Size : Positive) is
 record
 Count : Natural := 0; -- Number of items in the queue
 Front : Positive := 1; -- Pointer to first item in queue
 Rear : Positive := Max_Size; -- Pointer to last item in queue
 Items : Queue_Array (1..Max_Size); -- The element array
 end record;

end Queue;
```

Notice the initial values assigned to the three scalar fields in the record. The initial value assigned to `Rear` ensures that the element for the first call to `Enqueue` is placed into the first location of the array. Other initial values are possible as long as `Rear` indicates the array slot just before `Front`.

Body 7-1 contains the code that implements our queue algorithms. Note that `Dequeue`, like the stack `Pop` operation, does not actually remove the value of the element from the array. The value that has just been dequeued still physically exists in the array. It no longer exists in the queue, however, and cannot be accessed because of the change in `Front`.

### Body 7-1— Array-Based FIFO Queue Implements Specification 7-1

```
package body Queue is

procedure Enqueue (Queue : in out Queue_Type;
 Item : in Element_Type) is
begin
 if Queue.Count = Queue.Max_Size then
 raise OVERFLOW;
 else
 Queue.Rear := Queue.Rear rem Queue.Max_Size + 1;
 Queue.Items(Queue.Rear) := Item;
 Queue.Count := Queue.Count + 1;
 end if;
end Enqueue;
```

```

procedure Dequeue (Queue : in out Queue_Type;
 Item : out Element_Type) is
begin
 if Queue.Count = 0 then
 raise UNDERFLOW;
 else
 Item := Queue.Items(Queue.Front);
 Queue.Front := Queue.Front rem Queue.Max_Size + 1;
 Queue.Count := Queue.Count - 1;
 end if;
end Dequeue;

function Full (Queue : in Queue_Type) return Boolean is
begin
 return Queue.Count = Queue.Max_Size;
end Full;

function Empty (Queue : in Queue_Type) return Boolean is
begin
 return Queue.Count = 0;
end Empty;

procedure Clear (Queue : in out Queue_Type) is
begin
 Queue.Count := 0;
 Queue.Front := 1;
 Queue.Rear := Queue.Max_Size;
end Clear;

end Queue;

```

### ***Comparing Array Implementations***

The solution implemented in the above package is not nearly so simple or intuitive as our first queue design. What did we gain by adding some amount of complexity to our design? By using a more efficient Dequeue algorithm, we achieved better performance. To find out how much better, let's analyze the first design: Because the amount of work to move all the remaining elements is proportional to the number of elements, this version of Dequeue is an  $O(N)$  operation. The second array-based queue design only requires Dequeue to change the values of the Front indicator and to increment Count. The amount of work never exceeds some fixed constant, no matter how many elements are in the queue, so the algorithm is  $O(1)$ .

All of the other operations, for both array-based designs, are  $O(1)$ . No matter how many elements are in the queue, they do (essentially) a constant amount of work. That does not mean that the two Enqueue operations are the same in terms of lines of code or time of execution. It only means that the amount of work done by both of them does not depend on the number of items in a queue.

### *The Implementation of a Queue As a Linked Structure*

The major weakness of the array-based implementation is the need to declare an array big enough for a structure of the maximum expected size. This size is set when the queue object is elaborated. If a much smaller number of elements is needed, we have wasted a lot of space. If a larger number of elements is unexpectedly needed, we are in trouble. We cannot extend the size of the array after it has been elaborated.

We know, however, from our discussion of stacks in Chapter 6, that we can get around this problem by using dynamic storage allocation to get space for queue elements at run time. This implementation relies on the idea of linking the elements one to the next to form a chain.

In the array-based implementation of a queue, we decided to keep track of two indexes that pointed to the front and rear boundaries of the data in the queue. In a linked representation we can use two *access variables* to mark the front and the rear of the queue. The user's data is stored in dynamically allocated "nodes," along with a pointer to the following node in the queue. Here are the declarations for the private part of Specification 7-1 (page 270) using a linked implementation:

```
private

 type Node_Type; -- Incomplete type declaration

 type Node_Ptr is access Node_Type;

 type Node_Type is
 record
 Info : Element_Type; -- One item
 Next : Node_Ptr; -- Designates next node
 end record;

 type Queue_Type is
 record
 Front : Node_Ptr; -- Designates first item in the queue
 Rear : Node_Ptr; -- Designates last item in the queue
 end record;

end Queue;
```

Queue\_Type now consists of a record with two fields that designate nodes. The values in the queue are stored in nodes outside of this record. Because Queue\_Type is no longer a discriminated record, the Max\_size discriminant in the public portion of the package declaration given earlier in the chapter must be removed (programming problem 1 asks you to implement a queue that is completely compatible with the original specification). Figure 7-8 shows an example of a linked queue containing four elements. Because by now you realize that dynamically allocated nodes in linked structures exist somewhere in memory, rather than in adjacent locations like array slots, we are going to show the nodes arranged linearly for clarity.

Figure 7-8 shows a linked structure containing several nodes. Pictures like this are important aids in the design of algorithms that manipulate the structure. Many linked structures require our algorithms to handle special cases. Two common special cases are adding an element to an empty structure and deleting the last element from a structure.

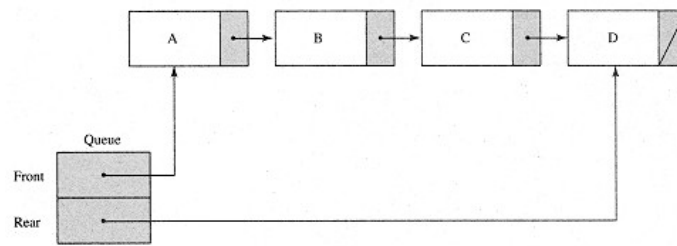


Figure 7-8  
A Linked Queue Representation

When developing our algorithms, it is therefore useful to draw pictures of the linked structure when it is empty and when it contains a single element. Figure 7-9 shows what our queue looks like when it is empty. Figure 7-10 shows what it looks like with a single element:

Our queue is empty when there are no nodes in the queue structure (Figure 7-9). To test for an empty queue, we need only check if `Queue.Front` is null. An alternative method is to check to see if `Queue.Rear` is null.

The Full operation for a linked queue is identical to the one we used for a linked stack. We try to allocate new memory and see if `STORAGE_ERROR` is raised. If it is raised, the queue is full. Otherwise the queue isn't full and we can deallocate the memory just obtained.

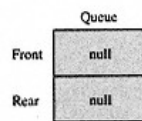


Figure 7-9  
An Empty Queue

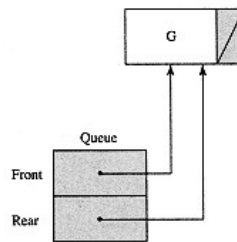


Figure 7-10  
A Queue Containing One Element

We can Dequeue elements from the queue using an algorithm similar to our stack Pop algorithm, with `Queue.Front` designating the first node in the queue. Figure 7-11 illustrates this algorithm for a queue containing more than one element. As in Pop, we need to keep a local access variable to designate the node being removed so its memory may be recycled.

Notice that the only access values changed as a result of the Dequeue operation are `Queue.Front` and the temporary variable `To_Recycle` used to designate the node whose memory is recycled. Are these two changes adequate when we dequeue an element in a queue that contains only one element as shown in Figure 7-10? No! Once we have

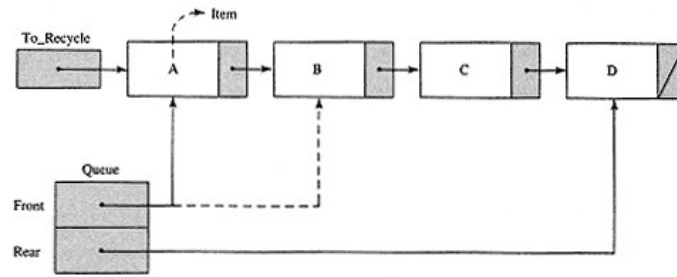


Figure 7-11  
The Dequeue Operation

recycled the last node, `Queue.Rear` no longer contains a valid access value. When we dequeue the last element, we must also be sure to set `Queue.Rear` to `null`. Here is the necessary code for the Dequeue operation. It includes a check for removing the last element from the queue.

```

Item := Queue.Front.all.Info; -- Get value from the front node
To_Recycle := Queue.Front; -- Save access to old front
Queue.Front := Queue.Front.all.Next; -- Change the front
Free (To_Recycle); -- Recycle the memory
if Queue.Front = null then -- Is the queue now empty?
 Queue.Rear := null; -- Set Rear to null as well
end if;

```

We can clear the queue using an algorithm similar to our stack Clear algorithm, with `Queue.Front` designating the first node in the queue. This algorithm contains a loop that recycled all the nodes. The loop terminates when `Queue.Front` becomes `null`. To match our picture of an empty queue, we then must set `Queue.Rear` to `null`.

We add new elements to the queue by inserting after the last node, as shown in Figure 7-12; we need a new Enqueue algorithm. Figure 7-12 shows that memory for a new node must be obtained and two access values changed to link this new node into our queue structure. We can accomplish these changes with the following assignment statement:

```

Queue.Rear.all.Next := new Node_Type' (Info => Item, Next => null);

```

Allocates memory for the new node.

Assign the new Item to Info.

There are no more nodes after this new one.

The Next field in the original rear node designates the new node. The new node is linked on the end.



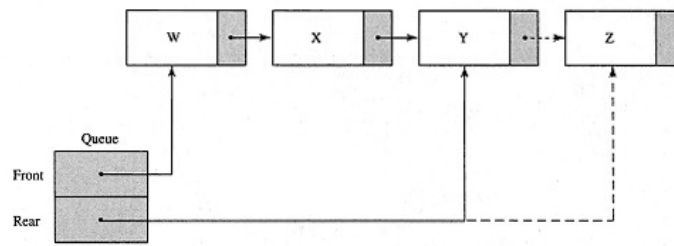


Figure 7-12  
The Enqueue Operation. Dashed Arrows Show the New Values of Pointers after the Enqueue Operation is Complete

By assigning the result of the allocation operator `new` to the `Next` field of the original rear node, we link the new node after it. We need a second assignment statement to change `Queue.Rear` so that it designates our newly added node.

```
Queue.Rear := Queue.Rear.all.Next;
```

Are these two assignment statements adequate for adding an element to an empty queue? Again the answer is no. The first assignment statement raises `CONSTRAINT_ERROR` because `Queue.Rear` is `null`—it is not designating a node. Adding an element to an empty queue is a special case. This case is incorporated into the following code for the `Enqueue` operation.

```
if Queue.Front = null then -- Is the queue empty?
 -- Yes, Front and Back should both designate the new node.
 Queue.Front := new Node_Type'(Info => Item, Next => null);
 Queue.Rear := Queue.Front;
else
 -- No, link a new node to the rear of the existing queue.
 Queue.Rear.all.Next := new Node_Type'(Info => Item, Next => null);
 Queue.Rear := Queue.Rear.all.Next;
end if;
```

Our initial picture of the linked queue structure (Figure 7-8) showed that the nodes were linked from `Front` to `Rear`. What if they had been reversed as in Figure 7-13? We then could use an algorithm similar to our `stack Push` algorithm for the `Enqueue` operation. But how could we `Dequeue`? To delete the last node of the linked queue, we need to be able to reset `Queue.Front` to designate the node preceding the deleted node. Because our access values all point forward, we can't get back to the preceding node. To accomplish this task, we either would have to start at the rear of the queue and traverse the whole list (an  $O(N)$  solution—very inefficient, especially if the queue is long) or else keep a linked structure with access values that point in both directions. Use of this kind of *doubly linked* structure (which we discuss in Chapter 9) is not necessary if we set up our queue access variables correctly to begin with.

### Linked Queue Package Body

Body 7-2 uses the linked structure illustrated in Figure 7-8 to implement Specification 7-1. As we discussed in Chapter 6, the `.all` in expressions that dereference record fields may be left out.

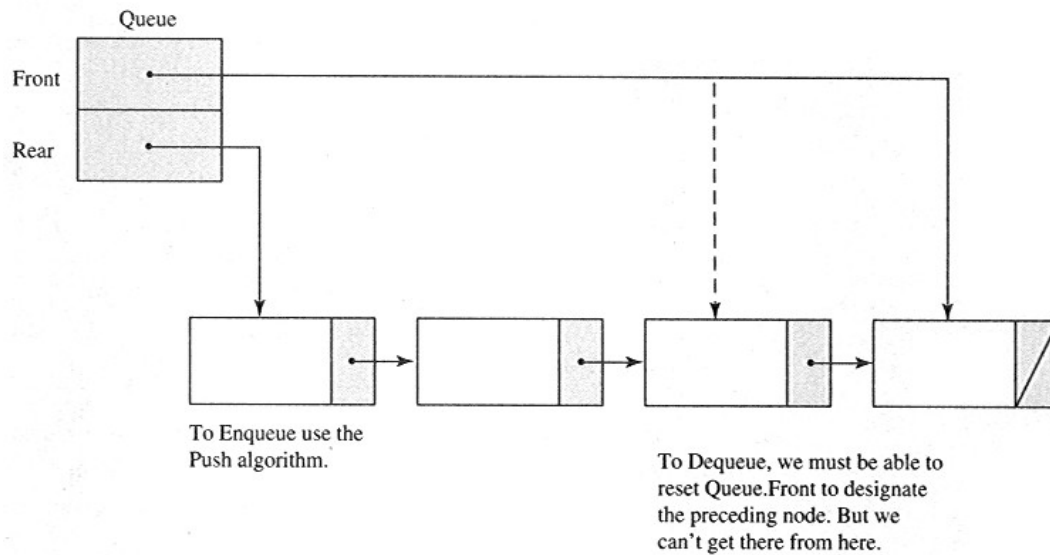


Figure 7-13  
A Bad Queue Design

## Body 7-2— Linked FIFO Queue Implements Specification 7-1

```
with Unchecked_Deallocation;
package body Queue is
```

```
-- Instantiate a procedure to recycle node memory
procedure Free is new Unchecked_Deallocation (Object => Node_Type,
 Name => Node_Ptr);
```

```

procedure Clear (Queue : in out Queue_Type) is
```

```
 To_Recycle : Node_Ptr; -- For recycling nodes
```

```
begin
 loop
 exit when Queue.Front = null; -- Exit when queue is empty
 To_Recycle := Queue.Front; -- Unlink the
 Queue.Front := Queue.Front.all.Next; -- front node
 Free (To_Recycle); -- Recycle the node
 end loop;
 Queue.Rear := null; -- Clean up Rear pointer
end Clear;
```

```

procedure Enqueue (Queue : in out Queue_Type;
 Item : in Element_Type) is
```

```
begin
 if Queue.Front = null then -- Is the queue empty?
 -- Yes, Front and Back should both designate the new node.
 Queue.Front := new Node_Type'(Info => Item, Next => null);
 Queue.Rear := Queue.Front;
```

```

else
 -- No, link a new node to the rear of the existing queue.
 Queue.Rear.all.Next := new Node_Type'(Info => Item, Next => null);
 Queue.Rear := Queue.Rear.all.Next;
end if;
exception
 when STORAGE_ERROR =>
 raise OVERFLOW;
end Enqueue;

procedure Dequeue (Queue : in out Queue_Type;
 Item : out Element_Type) is

 To_Recycle : Node_Ptr; -- For recycling nodes

begin
 if Empty (Queue) then
 raise UNDERFLOW;
 end if;
 Item := Queue.Front.all.Info; -- Get the value from the front node
 To_Recycle := Queue.Front; -- Save access to old front
 Queue.Front := Queue.Front.all.Next; -- Change the front
 Free (To_Recycle); -- Recycle the memory
 if Queue.Front = null then -- Is the queue now empty?
 Queue.Rear := null; -- Set Rear to null as well
 end if;
end Dequeue;

function Full (Queue : in Queue_Type) return Boolean is

 New_Node : Node_Ptr;

begin
 New_Node := new Node_Type; -- Get memory for a new node
 Free (New_Node); -- Recycle the memory
 return False; -- There is still memory available
exception
 when STORAGE_ERROR => -- When there is no memory available
 return True; -- the queue is full
end Full;

function Empty (Queue : in Queue_Type) return Boolean is
begin
 return Queue.Front = null;
end Empty;

end Queue;

```

### A Circular Linked Queue Design

Our `Queue_Type` record contains two pointers, one to each end of the queue. This design is based on the linear structure of the linked queue: Given only an access value that points to the front of the queue, we could follow the pointers to get to the rear, but this technique makes accessing the rear (to enqueue) an  $O(N)$  operation. With an access value that points to the rear of the queue, we could not access the front because the access values only point from the front to rear.

However, we could access both ends of the queue from a single pointer, if we made the queue *circularly linked*. That is, the `Next` field of the rear node would point to the front node of the queue (see Figure 7-14). Now `Queue_Type` simply is an access variable that points to a single queue node, rather than a record containing two access values. `Queue`, the external pointer to the queue, points to the rear node.

One interesting thing about this queue implementation is that it differs from the logical picture of a queue as a linear structure with two ends. This queue is a circular structure with no ends. What makes it a queue is its support of FIFO access.

In order to enqueue, we access the rear node directly through the access variable `Queue`. To dequeue, we must access the front node of the queue. We don't have an external access value pointing to this node, but we do have one pointing to the node preceding it—`Queue`. The access value to the front node of the queue is in `Queue.all.Next`. An empty queue would be represented by `Queue = null`.

Designing and coding the queue operations using a circular linked implementation is left as an exercise.

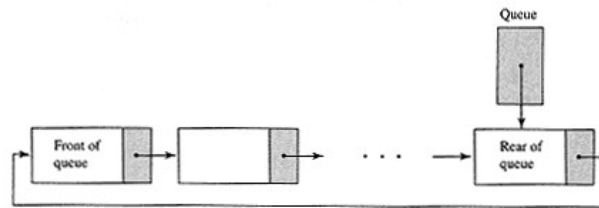


Figure 7-14  
A Circular Linked Queue

### Comparing the Queue Implementations

We now have looked at several different implementations of the FIFO Queue class. How do they compare? As we compared Stack implementations in Chapter 6, we look at several different factors: the length of the source code required to implement the solutions, the amount of memory required to store the structure, and the amount of "work" the solution requires as expressed in Big-O notation. We compare the two implementations that are completely coded in this chapter.

The size of the source code of the private part declarations, procedures, and functions for each implementation is summarized in the table that follows. As you can see from the table, the array-based implementation is somewhat smaller (nearly 30% fewer lines of code) than the linked implementation. There are fewer lines of code to develop and to maintain in the array implementation. Now that you are more familiar with Ada access type syntax and use, you may find that the linked implementation of a queue is more intuitive than the array implementation. Deciding which design is easier to understand is very subjective, but it is often a factor in selecting one approach over another.

### Comparison of Executable Source Lines of Code

|                             | <i>Array<br/>Implementation</i> | <i>Linked<br/>Implementation</i> |
|-----------------------------|---------------------------------|----------------------------------|
| Declarations (Private Part) | 8                               | 12                               |
| Clear                       | 6                               | 11                               |
| Empty                       | 4                               | 4                                |
| Full                        | 4                               | 10                               |
| Enqueue                     | 11                              | 11                               |
| Dequeue                     | 11                              | 12                               |
| Total                       | 43                              | 60                               |

Now let's compare the storage requirements. An array variable takes the same amount of memory, no matter how many array slots actually are used; we need to reserve space for the maximum possible number of elements. The linked implementation using dynamically allocated storage space only requires space for the number of elements actually in the queue at run time, plus space for the external access variable(s). Note, however, that the node elements are larger, because we must store the link (the `Next` field) as well as the user's data.

Let's see how these implementations compare if the queue contains strings (each requiring, say, 80 bytes). The maximum number of queue elements is 100 strings (`Max_Size = 100`). On our sample system, an `Integer` takes 2 bytes and an access value takes 4 bytes. The storage requirements of the array-based implementation are

$$(80 \text{ bytes/string} * 100 \text{ strings}) + (2 \text{ bytes/integer} * 3 \text{ integers}) = 8006 \text{ bytes}$$

no matter how many elements are in the queue at any time. The linked queue implementation requires

$$80 \text{ bytes/node (the string)} + 4 \text{ bytes/node (the "next" pointer)} = 84 \text{ bytes/node}$$

plus 8 bytes for the two external `Queue` pointers. The storage requirements of this queue are graphed in Figure 7-15(a). Note that the linked implementation does not always take less space than the array; when the number of elements in the queue exceeds 95, the linked queue requires more memory, due to the need to store the pointers.

If the queue element type were small, like a character or an integer, the pointer field could be larger than the user's data field. In this case, the space used by the linked representation exceeds that of the array-based representation much faster. Consider a queue that may contain up to 100 integer elements (2 bytes each). The storage requirements for the array-based queue are

$$(2 \text{ bytes/integer} * 100 \text{ integers}) + (2 \text{ bytes/integer} * 3 \text{ integers}) = 206 \text{ bytes}$$

no matter how many elements are in the queue at any time. The linked queue implementation requires

$$2 \text{ bytes/node (the Info field)} + 4 \text{ bytes/node (the Next field)} = 6 \text{ bytes/node}$$

plus 8 bytes for the two external `Queue` pointers. The storage requirements for this queue are graphed in Figure 7-15(b). When the number of elements in this queue exceeds 33, the

linked queue requires more memory, due to the need to store pointers that are twice as big as the queue element type!

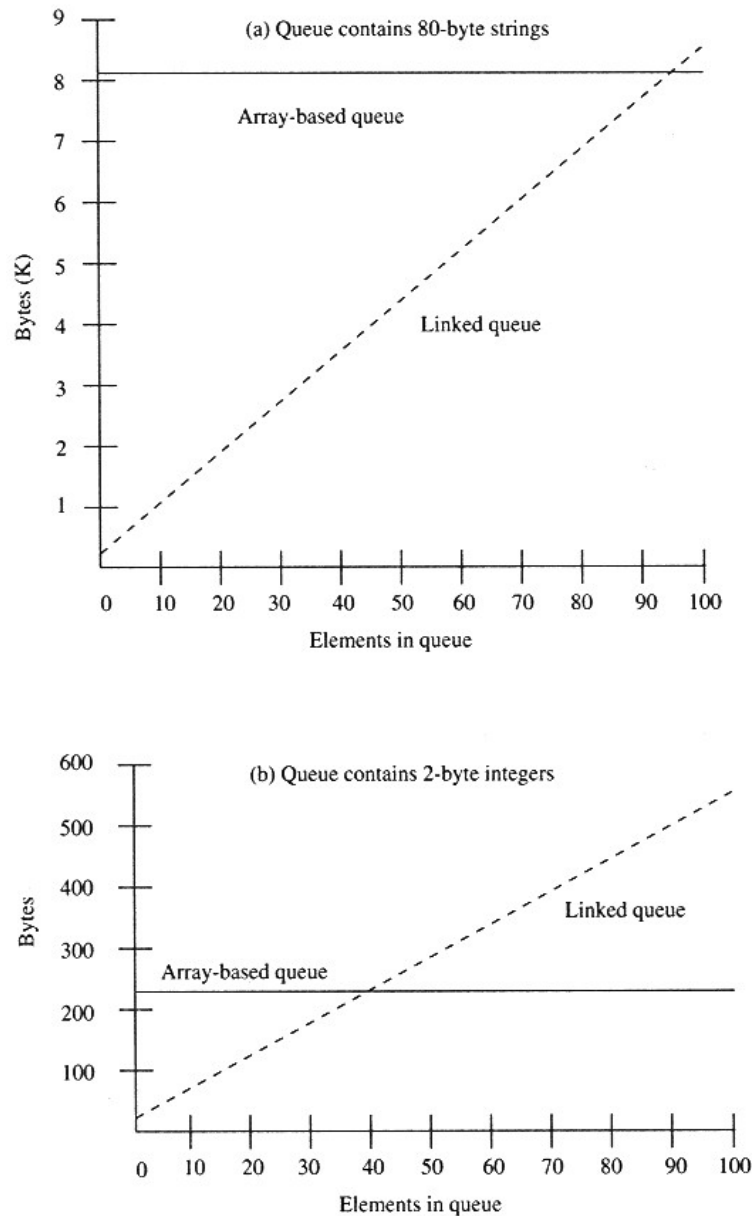


Figure 7-15  
Comparison of Storage Requirements

We also can compare the relative efficiency of the two implementations, in terms of Big-O. In both implementations, the `Full` and `Empty` operations are clearly  $O(1)$ . They always take a constant amount of work. What about `Enqueue` and `Dequeue`? Does the number of elements in the queue affect the amount of work performed by these operations? No, it does not; in both implementations, we can directly access the front and rear of the queue. The amount of work done by these operations is independent of the queue size, so these operations are also  $O(1)$ . Only the `Clear` operation differs from one implementation to the other. The array-based implementation merely sets the `Front` and `Rear` indexes, so it is clearly an  $O(1)$  operation. The linked implementation must process every node in the queue in order to free the node space. This operation, therefore, is  $O(N)$ , where  $N$  is the number of nodes in the queue. As with the array-based and linked implementations of

stacks, these two queue implementations are roughly equivalent in terms of the amount of work they do, only differing in one of the six operations. The following table summarizes the Big-O comparison of the queue operations:

| <b>Big-O Comparison of Queue Operations</b> |                                 |                                  |
|---------------------------------------------|---------------------------------|----------------------------------|
|                                             | <i>Array<br/>Implementation</i> | <i>Linked<br/>Implementation</i> |
| Clear                                       | <b>O(1)</b>                     | <b>O(N)</b>                      |
| Empty                                       | O(1)                            | O(1)                             |
| Full                                        | O(1)                            | O(1)                             |
| Enqueue                                     | O(1)                            | O(1)                             |
| Dequeue                                     | O(1)                            | O(1)                             |

Finally, let's look at an actual timed test case, with real numbers. The following table shows how long it takes *one particular computer* to enqueue and dequeue 100,000 integer elements. We also enqueued 100,000 integer elements onto a queue and timed how long it takes the clear operation to empty the queue.

| <b>Results (in seconds) of the Timed Implementation Comparisons</b> |                                 |                                  |
|---------------------------------------------------------------------|---------------------------------|----------------------------------|
|                                                                     | <i>Array<br/>Implementation</i> | <i>Linked<br/>Implementation</i> |
| Enqueue                                                             | 0.44                            | 0.82                             |
| Dequeue                                                             | 0.43                            | 0.68                             |
| Clear                                                               | 0.00*                           | 0.49                             |

Remember that the results of timing tests like these are dependent on the compiler used to convert the Ada program to machine language and on the computer on which the tests are run. As with the stack, the amount of time to clear an array-based queue is significantly less than the time required to clear a linked queue.

So which is better? Go back to the comparison of stack implementations in Chapter 6 and read the bottom line. The answer is the same for the two queue implementations: It depends on the requirements of your application. Some of the kinds of issues to consider when choosing an implementation are listed in the next table:

| <b>Considerations for Selecting a Queue Implementation</b>                                 |                                       |
|--------------------------------------------------------------------------------------------|---------------------------------------|
| <i>Situation</i>                                                                           | <i>Recommended<br/>Implementation</i> |
| The number of elements in the queue varies greatly from one program execution to the next. | Linked with access types              |
| The maximum number of elements cannot be predicted.                                        | Linked with access types              |

*(table continues on next page)*

---

\*Too small to measure

### Considerations for Selecting a Queue Implementation (cont.)

| <i>Situation</i>                                                                       | <i>Recommended Implementation</i> |
|----------------------------------------------------------------------------------------|-----------------------------------|
| The maximum number of elements is known, and the usual number is close to the maximum. | Array                             |
| The maximum number of elements is known and is small.                                  | Array                             |
| The maximum number of elements is large, and the element size is large.                | Linked with access types          |
| The system or programming language does not support dynamic storage allocation.        | Array                             |
| The program requirements specify a static implementation.                              | Array                             |
| The program requirements specify the use of dynamic allocation.                        | Linked with access types          |

### Testing the Queue Operations

You can test the FIFO queue class independently of the program that will use it with a *test driver*. Although it is possible to write a complicated interactive test driver that allows the tester to specify actions (such as enqueue and dequeue), the simplest test driver merely sets up a scenario that executes the various test cases. To make sure that you have tested all the necessary cases, make a test plan, listing all the queue operations and the tests needed for each. (For example, to test function `Empty`, you must call it at least twice—once when the queue is empty and once when it is not.)

We want to enqueue elements until the (array-based) queue is full; and then we need to call functions `Full` and `Empty` to see whether they correctly judge the state of the queue. Next we can dequeue all the elements in the queue, printing them out as we go, to make sure that they are correctly dequeued. At this point we can call the queue status functions again to see if the empty condition is correctly detected.

We also want to test out the tricky part of the array-based algorithm: Enqueue until the queue is full, dequeue an element, then enqueue again, forcing the operation to circle back to the beginning of the array. At each point in the test, you should print out text explaining what the test driver is doing.

We can use the same test driver for testing both queue implementations: We merely name the appropriate generic package in a `with` clause and instantiate an appropriate new package. Note that we get different results for the `Full` function, depending on the implementation.

Because the type of data stored in the queue has no effect on the operations that manipulate the queue, you can choose `Element_Type` to simplify the test driver. Suppose that the queue in your application program contains a maximum of 1000 record-type elements, each with many fields (a queue of student information records, for instance). You don't need to enqueue actual student records, because the queue element type is not relevant to how `Enqueue` works. For the test driver, you can declare `Element_Type` to be `Character` to simplify setting up the test. You also don't need to test a queue of 1000 elements. Knowing that the code works the same whether `Max_Size` is 10 or 1000, you can set `Max_Size` to 10 for the test.



## Summary

In this chapter we examine the definition and operations of a FIFO queue. We also discuss some of the design considerations we encounter when we use an array to contain the elements of a queue. Though the array itself is a random-access structure, our logical view of the queue as a FIFO structure limits us to accessing only the elements in the front and rear positions of the queue stored in the array. As with stacks, the problem with a static array-based implementation is the need to determine the maximum queue size before compilation. The queue is logically a dynamic structure, but the array is physically a static structure.

The linked implementation of a FIFO queue allows us to allocate space for queue elements as needed. Now that you are getting used to Ada's access types, this implementation is becoming more intuitive—perhaps even more intuitive than the array implementation that we have developed.

There usually is more than one functionally correct design for the same data structure. When multiple correct solutions exist, the requirements and specifications of the problem may determine which solution is the best design.

In the design of data structures and algorithms, we find that there are often trade-offs. A more complex algorithm may result in more efficient execution; a solution that takes longer to execute may save memory space. As always, we must base our design decisions on what we know about the problem's requirements.

## Exercises

1. Describe the accessing functions of a FIFO queue at the abstract level.
2. Show what is written by the following segments of code, given that `Queue` is a FIFO queue of integers; `Stack` is a stack of integers; and `X`, `Y`, and `Z` are integer variables.

```
(a) X := 0;
 Y := 1;
 Enqueue (Queue, X);
 Enqueue (Queue, Y);
 Dequeue (Queue, Y);
 Z := Y + 5;
 Enqueue (Queue, Z);
 Enqueue (Queue, 7);
 Dequeue (Queue, X);
 Enqueue (Queue, Y);
loop
 exit when Empty (Queue);
 Dequeue (Queue, Z);
 Integer_IO.Put(Z);
 Text_IO.New_Line;
end loop;
```

```

(b) X := 0;
 Y := i;
 Z := X + Y;
 loop
 exit when Z >= 10;
 if Z rem 2 = 0 then
 Push (Stack, Z);
 else
 Enqueue (Queue, Z);
 end if;
 X := Y;
 Y := Z;
 Z := X + Y;
 end loop;

 Text_IO.Put ("Stack contains: ");
 loop
 exit when Empty (Stack);
 Pop (Stack, Z);
 Integer_IO.Put (Z);
 end loop;
 Text_IO.New_Line;

 Text_IO.Put ("Queue contains: ");
 loop
 exit when Empty (Queue);
 Dequeue (Queue, Z);
 Integer_IO.Put (Z);
 end loop;
 Text_IO.New_Line;

```

3. You are given the following declarations:

```

subtype Name_String is String (1..20);

type Name_Rec is
 record
 First : Name_String;
 Last : Name_String;
 end record;

```

(a) Instantiate a queue package whose elements are Name\_Rec.

(b) Using the package you just instantiated, declare a queue object that holds a maximum of 250 names.

4. Indicate whether each of the following applications would be suitable for a FIFO queue.

- (a) An ailing company wants to evaluate employee records in order to lay off some workers on the basis of service time (the most recently hired employees will be laid off first).
- (b) A program is to keep track of patients as they check into a clinic, assigning them to doctors on a first-come, first-served basis.
- (c) A program to solve a maze is to backtrack to an earlier position (the last place where a choice was made) when a dead-end position is reached.
- (d) An inventory of parts is to be processed by part number.
- (e) An operating system is to process requests for computer resources by allocating the resources in the order in which they are requested.

- (f) A grocery chain wants to run a simulation to see how average customer wait time would be affected by changing the number of checkout lines in the stores.
- (g) A dictionary of words used by a spelling checker is to be created.
- (h) Customers are to take numbers at a bakery and be served in order when their numbers come up.
- (i) Gamblers are to take numbers in the lottery and win if their numbers are picked.

5. True or False?

- (a) The comparison of the implementations of a FIFO queue in this chapter proves that there is always one implementation of a data structure that is clearly the best.
- (b) The statement `Deque (Queue, Q + 5)` would cause a run-time error.
- (c) All data structures that are called "queues" have the first in, first out accessing function.

Use the following instructions for Exercises 6–11:

Show the result of the given operations on the queues, using the final array-based implementation discussed in this chapter. If `OVERFLOW` or `UNDERFLOW` is raised, mark the appropriate spot; otherwise, show any changes to the queue. (Note that some of the values shown in the array may not be elements of the queue.)

6. Enqueue (Queue, 'J');

| Queue.Items |     |     |     |     |
|-------------|-----|-----|-----|-----|
| V           | W   | X   | Y   | Z   |
| (1)         | (2) | (3) | (4) | (5) |

Queue.Front = 1  
Queue.Rear = 4  
Queue.Count = 4

OVERFLOW \_\_\_\_\_ UNDERFLOW \_\_\_\_\_

| Queue.Items |     |     |     |     |
|-------------|-----|-----|-----|-----|
|             |     |     |     |     |
| (1)         | (2) | (3) | (4) | (5) |

Queue.Front = \_\_\_\_\_  
Queue.Rear = \_\_\_\_\_  
Queue.Count = \_\_\_\_\_

7. Enqueue (Queue, 'K');

| Queue.Items |     |     |     |     |
|-------------|-----|-----|-----|-----|
| V           | W   | X   | Y   | Z   |
| (1)         | (2) | (3) | (4) | (5) |

Queue.Front = 5  
Queue.Rear = 5  
Queue.Count = 1

OVERFLOW \_\_\_\_\_ UNDERFLOW \_\_\_\_\_

| Queue.Items |     |     |     |     |
|-------------|-----|-----|-----|-----|
|             |     |     |     |     |
| (1)         | (2) | (3) | (4) | (5) |

Queue.Front = \_\_\_\_\_  
Queue.Rear = \_\_\_\_\_  
Queue.Count = \_\_\_\_\_

8. Enqueue (Queue, 'L');

Queue.Items

|   |   |   |   |   |
|---|---|---|---|---|
| V | W | X | Y | Z |
|---|---|---|---|---|

(1)

(2)

(3)

(4)

(5)

Queue.Front = 4

Queue.Rear = 2

Queue.Count = 4

OVERFLOW

UNDERFLOW

Queue.Items

|  |  |  |  |  |
|--|--|--|--|--|
|  |  |  |  |  |
|--|--|--|--|--|

(1)

(2)

(3)

(4)

(5)

Queue.Front =

Queue.Rear =

Queue.Count =

9. Dequeue (Queue, Value);

Queue.Items

|   |   |   |   |   |
|---|---|---|---|---|
| V | W | X | Y | Z |
|---|---|---|---|---|

(1)

(2)

(3)

(4)

(5)

Queue.Front = 5

Queue.Rear = 5

Queue.Count = 1

OVERFLOW

UNDERFLOW

Value

Queue.Items

|  |  |  |  |  |
|--|--|--|--|--|
|  |  |  |  |  |
|--|--|--|--|--|

(1)

(2)

(3)

(4)

(5)

Queue.Front =

Queue.Rear =

Queue.Count =

10. Dequeue (Queue, Value);

Queue.Items

|   |   |   |   |   |
|---|---|---|---|---|
| V | W | X | Y | Z |
|---|---|---|---|---|

(1)

(2)

(3)

(4)

(5)

Queue.Front = 5

Queue.Rear = 3

Queue.Count = 4

OVERFLOW

UNDERFLOW

Value

Queue.Items

|  |  |  |  |  |
|--|--|--|--|--|
|  |  |  |  |  |
|--|--|--|--|--|

(1)

(2)

(3)

(4)

(5)

Queue.Front =

Queue.Rear =

Queue.Count =

## 11. Dequeue (Queue, Value);

| Queue.Items |     |     |     |     | Queue.Front = 3<br>Queue.Rear = 2<br>Queue.Count = 5 |
|-------------|-----|-----|-----|-----|------------------------------------------------------|
| V           | W   | X   | Y   | Z   |                                                      |
| (1)         | (2) | (3) | (4) | (5) |                                                      |

OVERFLOW \_\_\_\_\_ UNDERFLOW \_\_\_\_\_ Value \_\_\_\_\_

| Queue.Items |     |     |     |     | Queue.Front = _____<br>Queue.Rear = _____<br>Queue.Count = _____ |
|-------------|-----|-----|-----|-----|------------------------------------------------------------------|
|             |     |     |     |     |                                                                  |
| (1)         | (2) | (3) | (4) | (5) |                                                                  |

## 12. You are given the following procedure declaration:

```

procedure Enqueue (Queue : in out Queue_Type;
 New_Element : in Element_Type;
 Overflow : out Boolean);
-- This procedure adds New_Element to the rear of the Queue.

-- Preconditions : None
--
-- Postconditions : Overflow = (queue is already full)
-- IF NOT Overflow, Queue = original Queue with
-- New_Element added at the rear.
-- ELSE Queue = original Queue

```

(a) Write the body for procedure Enqueue using the array-based implementation discussed in the chapter.

(b) Write the body for procedure Enqueue using the linked implementation discussed in the chapter.

## 13. You are given the following procedure declaration:

```

procedure Dequeue (Queue : in out Queue_Type;
 Deq_Element : out Element_Type;
 Underflow : out Boolean);
-- This procedure removes the front element from Queue and
-- returns its value in Deq_Element.

-- Preconditions : None
--
-- Postconditions : Underflow = (queue is already empty)
-- IF NOT Underflow, Queue = original Queue with
-- front element removed AND Deq_Element = front
-- element from original queue.
-- ELSE Queue = original empty Queue

```

(a) Write procedure Dequeue using the array-based implementation discussed in the chapter.

(b) Write procedure Dequeue using the linked implementation discussed in the chapter.

Use the following declarations for Exercises 14–17:

Recall that the count field in the array-based queue discussed in the chapter allowed us to distinguish between a full queue and an empty queue. In the following design, a Boolean field is used for this purpose.

```

private

type Queue_Array is array (Positive range <>) of Element_Type;

type Queue_Type (Max_Size : Positive) is
 record
 Full : Boolean := False; -- True if queue is full
 Front : Positive := 1; -- Pointer to first item in queue
 Rear : Positive := Max_Size; -- Pointer to last item in queue
 Items : Queue_Array (1..Max_Size); -- The element array
 end record;

end Queue;

```

14. Write the `Full` and `Empty` functions that go with the declarations above using the specification given by FIFO queue package declaration in this chapter.

15. Write the `Enqueue` and `Dequeue` procedures that go with the declarations above using the specification given by FIFO queue package declaration in this chapter.

16. You are given the following function declaration:

```

function Count (Queue : in Queue_Type) return Natural;
-- Returns the number of elements in Queue

-- Preconditions : None
--
-- Postconditions : Returns number of elements in Queue

```

Write the body for function `Count` using the above declarations for `Queue_Type`. What is the Big-O of this function?

17. Write the `Clear` procedure that goes with the previous declarations using the specification given by the FIFO queue package declaration in this chapter. How does this implementation compare to the equivalent array-based queue operation in the chapter, using Big-O notation?

18. You are given the following function declaration:

```

function Count (Queue : in Queue_Type) return Natural;
-- Returns the number of elements in Queue

-- Preconditions : None
--
-- Postconditions : Returns number of elements in Queue

```

Write the body for function `Count` using the declarations for `Queue_Type` given for the *linked* implementation discussed in the chapter. What is the Big-O of this function?

19. The DCT hump yard application discussed in this chapter had track sections for cars going to Austin, Bethesda, and Plattsburgh. Cars bound to other destinations are left coupled to the end of a string of cars attached to the locomotive moving cars from the incoming track to the hump. What data structure is most suitable for modeling the string of cars attached to this locomotive?

20. How do you decide between using an exceptional handler for `OVERFLOW` or `UNDERFLOW` and using the functions `Full` and `Empty`?

21. A *deque* (pronounced "deck") is what you get when you cross a stack with a schizophrenic queue. You can add to and delete from either end of a deque. Maybe it should be called a FOLIFOLO (First Or Last In, First Or Last Out) structure. Using an array to contain the elements in the deque, write the following:

- (a) The declarations that you would use to implement `Deque_Type`. Be sure to include the necessary initial values.
- (b) The declaration (including all necessary comments to specify the operation) and body of procedure `In_Deque_Front` (add element to front).
- (c) The declaration (including all necessary comments to specify the operation) and body of procedure `In_Deque_Rear` (add element to rear).
- (d) The declaration (including all necessary comments to specify the operation) and body of procedure `Out_Deque_Front` (remove element from front).
- (e) The declaration (including all necessary comments to specify the operation) and body of procedure `Out_Deque_Rear` (remove element from rear).

22. Repeat Exercise 21 using a *linked* deque representation.

23. A queue is kept in a record (`Queue_Data_Type`) that has four fields: `Elements` (an array that contains the queue elements); `Front` (the index of the array slot containing the front element); `Rear` (the index of the array slot containing the rear element); and `Count` (the number of elements currently in the queue). The record variable is not declared, but rather it is allocated dynamically when a queue is needed. Here are the declarations for this data structure:

```
type Queue_Array is array (Positive range <>) of Element_Type;
```

```
type Queue_Data_Type (Max_Size : Positive) is
 record
 Count : Natural := 0;
 Front : Positive := 1;
 Rear : Positive := Max_Size;
 Items : Queue_Array (1..Max_Size);
 end record;
```

```
type Queue_Type is access Queue_Data_Type;
```

```
Queue : Queue_Type; -- A queue of elements
```

- (a) Is the space for this array-based data structure statically or dynamically allocated?
- (b) How much space is reserved for the variable `Queue` at the beginning of the program's execution?
- (c) Write the body for the following declaration:

```
procedure Create_Queue (Queue : in out Queue_Type;
 Max_Size : in Positive);
-- Brings a new queue into being

-- Preconditions : The Queue must not have been
-- already created.
--
-- Postconditions : The Queue is created and empty.
```

- (d) Is a `Full` operation for this representation meaningful? If yes, write a body for the `Full` operation. If the operation is not meaningful for this implementation, write a stub.
- (e) If you could write the `Enqueue` and `Dequeue` procedures by modifying the source code of one of the queue implementations in the chapter, which one would you choose to modify?

24. You are given the following procedure declarations:

```
procedure Print_Element (Element : in Element_Type);
-- Prints the given element
```

```

procedure Print_Queue (Queue : in Queue_Type);
-- Prints all the elements in Queue from front to back.

-- Preconditions : None
--
-- Postconditions : The elements in the queue are printed.
-- The Queue is unchanged.

```

- (a) Write the body for procedure `Print_Queue` as an operation provided as part of the FIFO Queue ADT, using the array-based queue implementation from the chapter. You should call procedure `Print_Element` but don't write the body of this procedure.
- (b) Write the body for procedure `Print_Queue` as an operation provided as part of the FIFO Queue ADT, using the linked queue implementation from the chapter. You should call procedure `Print_Element` but don't write the body of this procedure.
- (c) Write the body for procedure `Print_Queue` using only the operations provided in the FIFO Queue ADT Specification, without any knowledge of how the queue is implemented. What changes must be made to the mode of the parameter `Queue`? You should call procedure `Print_Element` but don't write the body of this procedure.
- (d) Do the procedures in (a), (b), and (c) differ in efficiency in terms of execution speed? In terms of Big-O notation?
25. A queue is implemented as a circular linked structure, as described in the chapter, with the external pointer accessing the "rear" element.
- (a) Draw a sketch of a queue with one node.
- (b) Draw a sketch of an empty queue.
- (c) Implement the `Clear` and `Empty` operations.
- (d) Implement the `Enqueue` and `Dequeue` operations.
- (e) Does this implementation change the storage requirements of the queue, as compared with the linear linked queue? Does it change the Big-O approximations of the work required to do each operation?
26. A queue contains 12-byte elements. Assume that an access value takes 4 bytes and an integer takes 2 bytes, and that the maximum queue size is 80 elements.
- (a) Fill in the chart showing the storage requirements, using the implementations developed in the chapter.

| <i>Number of<br/>Elements</i> | <i>Array-Based<br/>Implementation</i> | <i>Linked<br/>Implementation</i> |
|-------------------------------|---------------------------------------|----------------------------------|
| 0                             | _____                                 | _____                            |
| 10                            | _____                                 | _____                            |
| 30                            | _____                                 | _____                            |
| 50                            | _____                                 | _____                            |
| 80                            | _____                                 | _____                            |

- (b) At what point does the linked implementation take more space than the array-based implementation?

Use the following information for Exercises 27–29:

A particular operating system queues the user jobs that are waiting to be executed according to the following scheme:



- System users have relative priorities based on their user ID numbers:

```

Users 0-99 Highest (for example, company executives)
Users 100-199 Next to highest (for example, executive secretaries)
Users 200-299 Next highest (for example, technical leaders)
.
.
.
Users 800-899 Next to lowest (for example, regular programmers)
Users 900-999 Lowest (those whose jobs only run at 3 a.m. on weekends)

```

- Within each priority group the jobs execute in the order in which they arrive in the system.
- If there is a highest priority job queued, it executes before any other job; if not, if there is a next to highest priority job queued, it runs before any lower priority jobs; and so on. That is, a lower priority job runs only when there are no higher priority jobs waiting.
- The system has an array of FIFO queues to hold the queues for the various priority levels:

```
type Job_Array is array (0..9) of Queue_Type;
```

```
Jobs : Job_Array -- queues of job "tokens" for each priority level
```

In completing the following exercises, you may call any of the queue operations specified in the FIFO queue package.

27. Write procedure `Add_Job`, which receives a user ID and a token (representing the job to be executed) and adds the token to the appropriate queue for that user's priority level.

28. Write procedure `Get_Next_Job`, which returns the token for the highest priority job that is queued for execution. (The token should be removed from the queue.)

29. The system is going down for maintenance. All jobs that are waiting to be executed have to be purged from the job queues. Fortunately this system is very friendly. It notifies users when their jobs are being canceled so that the users know to resubmit the jobs later. Procedure `Notify`, with `Token` and `Message`, as input, takes care of this notification. Write a procedure `Clean_Up_Jobs` that sends `Message` to all the users with queued jobs (call procedure `Notify` to send the message). The highest priority users should be notified first, of course.

### Programming Problems

- (a) Why is the linked queue implementation developed in this chapter inadequate for solving the DCT hump yard application?
  - Modify the declaration of `Queue_Type` in the package declaration for the linked queue implementation so that a linked queue may be used in the DCT hump yard application. Use the stack declarations given in programming problem 2 in Chapter 6 as a guide.
  - Modify the linked queue package to reflect the changes you made to `Queue_Type` in part (b).
  - Develop a formal test plan for testing the queue package you implemented in part (c).
  - Write a program to carry out the test plan you developed in part (d).
- The local medical clinic has decided to automate its scheduling services. You have been assigned to design the initial version of the schedules. The basic functions that the clinic has in mind are doctor check-in and check-out and patient check-in and check-out.

A doctor checks in by telling the scheduler his or her name, an examination room number, and a medical specialty code. Each doctor has a favorite room. The scheduler checks to see whether the room is free. If so, it assigns this doctor to the room; if not, it rejects the request with a message, and the doctor can try again to check in. When a doctor checks out, the examination room is freed.

A patient checking in gives a name, age, specialist code, and emergency indication. The scheduler tries to match up the patient with a doctor according to a set of rules that will be described below. If there is a match, the patient is seen by the assigned doctor. If this doctor is currently seeing a patient, the new patient is queued to see the doctor according to the emergency indicator. Usually there is no emergency, and the patient is put at the end of the doctor's waiting list; if there is an emergency, however, the patient is put at the front of the waiting list ahead of any other patients.

The rules for assigning doctors to patients are as follows:

1. Any patient under age 16 is assigned to see a pediatrician.
2. Patients age 16 and older are assigned a doctor according to the specialty requested. If there is no doctor in the clinic with the requested specialty, the patient is assigned to a general practitioner (GP). If there is no GP, the patient can be assigned to any doctor.
3. If there is more than one doctor of the requested specialty, the patient is assigned to the doctor with the shortest waiting list.

When a patient checks out, the doctor he or she was assigned to is available to see the next patient if there is anyone in the waiting list.

**Input**

Since this is an interactive system, your program should prompt the users to input the correct information. The initial prompt is

Type D for Doctor or P for Patient:

The next prompt is

Type I for checking in or O for checkout:

According to the request, your program should prompt the user for any other needed information, as indicated in the following table:

| Action            | Additional information                                                |
|-------------------|-----------------------------------------------------------------------|
| Doctor check-in   | Doctor's name<br>Room number<br>Specialty code                        |
| Doctor check-out  | Doctor's name                                                         |
| Patient check-in  | Patient's name<br>Age<br>Specialty (code requested)<br>Emergency flag |
| Patient check-out | Patient's name<br>Room number                                         |

You may define the format for the input processed by your program.

## Output

The output for each request is in the form of messages to the user, according to the request, as indicated in the following table:

| <i>Action</i>     | <i>Message</i>                                                                                                              |
|-------------------|-----------------------------------------------------------------------------------------------------------------------------|
| Doctor check-in   | Confirmation that room is available or error message if room is in use                                                      |
| Doctor check-out  | Goodbye message                                                                                                             |
| Patient check-in  | Message telling patient which room to go to and which doctor has been assigned. If no doctor available, apologetic message. |
| Patient check-out | Goodbye message. At a later time we may add billing information at this point.                                              |

In addition to printing the messages on the screen, you should also write the requests and messages to a transaction file (TRANSACTION.OUT), to be turned in with your program listing.

## Details and Assumptions

1. There are 100 examination rooms at the clinic, each with a waiting room attached.
2. Specialty codes are as follows:

|     |                       |
|-----|-----------------------|
| PED | Pediatrics            |
| GEN | General practice      |
| INT | Internal medicine     |
| CAR | Cardiology            |
| SUR | Surgeon               |
| OBS | Obstetrics            |
| PSY | Psychiatry            |
| NEU | Neurology             |
| ORT | Orthopedics           |
| DER | Dermatology           |
| OPT | Ophthalmology         |
| ENT | Ear, Nose, and Throat |

3. You may assume that no patient leaves without checking out. (That is, every doctor becomes free eventually.)
4. No one leaves before he or she sees the assigned doctor. (That is, no one has to be taken out of the waiting queue.) The clinic is open 24 hours a day, 7 days a week.
5. If a doctor checks out while there is still a waiting list of patients assigned to him or her, the patients must be reassigned to other doctors.

## Object Classes

Doctors and patients are classes in this problem that can be implemented as records. Define these record types in a definition package.

Examination rooms are major objects in this problem. Each examination room may have a doctor, a patient, and a waiting list. Since there are many examination rooms you should implement the examination room class as an ADT. You must determine what attributes and operations are appropriate for this class.

The clinic is a collection of examination rooms. The number of rooms is fixed, so you may use an array of records to represent it.

Because this chapter is about queues, it is the waiting list attached to each examination room that is of most interest to us. We have seen that patients are seen in the order in which they are added to the list (a simple queue) with one exception: Emergency patients are added at the front of the queue. What we have here is a special data structure, kind of a *deque* (described in Exercise 21) in which additions may be made to either end, but deletions are always made at the front of the list. You must determine what attributes and operations are appropriate for this class.

***Turn In***

1. listing of the program
2. listings of all packages
3. hard copy of TRANSACT.OUT file

**8—****Linear Lists****GOALS**

After reading this chapter, you should be able to

- describe the structure of a list and its operations at the abstract level
- describe how an ordered list may be implemented as a sequential or a linked structure
- implement the following list operations for both sequential and linked implementations:
  - determine whether the list is empty or full
  - clear a list
  - insert an element
  - retrieve an element
  - modify an element
  - delete an element
  - traverse the list elements in order
- contrast the sequential and linked implementations of a list in terms of the Big-O approximations of their operations

We all know intuitively what a "list" is; in our everyday lives we use lists all the time—grocery lists, lists of things to do, lists of addresses, lists of party guests. In this chapter and the next two we look at the list class from our three perspectives.

## The Abstract Level

In computer programs, lists are very useful and common data structures. From a programming point of view, a list is a *homogeneous* collection of elements, with a linear relationship between the elements. This definition means that, at the abstract level, each element in the list except the first one has a unique predecessor and each element except the last one has a unique successor. (At the implementation level, there is also a relationship between the elements, but the physical relationship may not be the same as the logical one.) Lists can be unordered—their elements may be placed into the list in no particular order. Lists also can be ordered in different ways. For instance, stacks and queues are lists that are ordered according to the time when their elements were added.

---

### Sorted (Value-Ordered) List

A list in which the elements are ordered according to their value.

### Key

Field in a record whose value(s) are used to determine the logical (and/or physical) order of the records in a list.

### Key-Ordered List

A list in which the elements are records and are ordered according to the value of a key field of each element.

---

Lists also can be ordered *by value*; for instance, a list of strings can be ordered alphabetically, or a list of grades can be ordered numerically. Often a value-ordered list is called a **sorted list**. When the elements in a value-ordered list are records, rather than scalar data, their logical (and often physical) order is determined by one of the fields in the record, the record **key**. For example, we can order a list of students on the Dean's List alphabetically by name or numerically by student ID. In the first case, the Name field of the record is the key; in the second case, the ID field is the key. Such value-ordered lists also are called **key-ordered lists**.

If a list cannot contain records with duplicate keys, it is considered to have *unique keys*. (To generalize our list nomenclature, we can consider a list of scalar values to be a list whose elements are keys.) This chapter deals with lists of elements with unique keys, ordered from smallest to largest key value.

## Operator Classes

---

### Constructors

Operations that alter the state of an object.

### Observers

Operations that allow us to observe the state of an object without changing it.

### Iterators

Operations that allow us to process all of the components of a collection class object.

---

There are many different operations that programmers can provide for lists. For different applications we can imagine all kinds of things users might need to do to a list of elements. As a result there are no "standard" list operations. However, all list operations fall into three categories: **constructors**, **observers**, and **iterators**.

A constructor is an operation that alters the state of the list. The push, pop, and clear operations we developed for stacks in Chapter 6 are constructor operations. So are the enqueue, dequeue, and clear operations developed for queues in Chapter 7. Another common constructor is *create*, an operation to initialize an object. We have used initial default values in our record declarations so that create is done implicitly whenever we declare a variable of the abstract data type. Most other programming languages do not support initial default values in record declarations so an explicit create operation is required for all classes.

An observer is an operation that allows us to observe the state of an object without changing it. The functions `Full` and `Empty` developed for stacks and queues are observers.

An iterator is an operation that allows us to process all the components in a data structure in sequence. Iterators do not alter the data structure. However, they may alter individual elements within the data structure. An operation that prints all the elements in a queue is an iterator. So is an operation that adds 10 to the final exam score for each student in a list of students.

### *A List Package Declaration*

Specification 8-1 contains a small collection of basic list operations. The constructors are operations to add a new element, to delete an element, to modify an element, and to clear the list. The observers are operations to determine whether a list is empty or full, determine the number of elements in the list, and to retrieve a copy of an element given its key. We include a generic iterator from which we may instantiate as many different iterators as needed for our application. This generic procedure requires us to supply an operation (as a generic parameter) that it applies to each component in the list.

#### **Specification 8-1— Sorted List Class**

generic

```

type Element_Type is private; -- The type of element in the list
type Key_Type is limited private; -- The type of key in the element

-- The user must supply a function that returns the Key of an Element.
with function Key_Of (Element : in Element_Type) return Key_Type;
-- The user must supply functions for comparing Keys.
with function "=" (Left : in Key_Type; Right : in Key_Type) return Boolean;
with function "<" (Left : in Key_Type; Right : in Key_Type) return Boolean;

package Sorted_List is

-- This package implements an ordered list ADT.

-- Each element in the list has a unique key.

-- The elements in the list are ordered from smallest to largest using the "<"
-- function supplied as a generic parameter. The key value of the predecessor
-- of an element is less than the key value of the element, which is less than
-- the key value of the successor of the element.

type List_Type (Max_Size : Positive) is limited private;

DUPLICATE_KEY : exception;
KEY_ERROR : exception;
OVERFLOW : exception;

-- Constructors

procedure Clear (List : in out List_Type);
-- This procedure removes all elements from List.

-- Preconditions : None
--
-- Postconditions : List is empty.

```

```

procedure Insert (List : in out List_Type;
 Item : in Element_Type);
-- This procedure adds Item to List.

-- Preconditions : None
--
-- Postconditions : List = original list + Item
--
-- Exceptions : OVERFLOW If there is no room for Item. List is unchanged.
-- DUPLICATE_KEY If an element already exists in the list
-- with the same key as Item. List is unchanged.

procedure Delete (List : in out List_Type;
 Key : in Key_Type);
-- Deletes the element containing Key from List.

-- Preconditions : None
--
-- Postconditions : List = original list with Key removed
--
-- Exceptions : KEY_ERROR If there is no element with Key in List.
-- List is unchanged.

procedure Modify (List : in out List_Type;
 Element : in Element_Type);
-- Replace existing list element with same key as Element.

-- Preconditions : None
--
-- Postconditions : List = original list with value of Element replacing
-- like-keyed original element
--
-- Exceptions : KEY_ERROR If there is no element with the key of
-- Element in List. List is unchanged.

procedure Retrieve (List : in List_Type;
 Key : in Key_Type;
 Element : out Element_Type);
-- A copy of the element that contains Key is returned.

-- Preconditions : None
--
-- Postconditions : Element = copy of element containing Key
-- List is unchanged.
--
-- Exceptions : KEY_ERROR If there is no element with the
-- Key in List. List is unchanged.

```



```
-- Observers
```

```

function Full (List : in List_Type) return Boolean;
-- Determines whether List is full.
```

```
-- Preconditions : None
--
-- Postconditions : Full = (List is full)
```

```

function Empty (List : in List_Type) return Boolean;
-- Determines whether List is empty.
```

```
-- Preconditions : None
--
-- Postconditions : Empty = (List is empty)
```

```

function Length (List : in List_Type) return Natural;
-- Determines the number of elements in List.
```

```
-- Preconditions : None
--
-- Postconditions : Length = number of elements in List
```

```
-- Iterator
```

```

generic
 -- The user must supply a procedure that will be called once for every
 -- element in the list. Process may not change the key of Element.
 with procedure Process (Element : in out Element_Type);
```

```
procedure Traverse (List : in out List_Type);
-- Calls procedure Process once for every element in the list. Elements
-- are "processed" from smallest key to largest key.
```

```
-- Preconditions : None
--
-- Postconditions : For very element in List, Process(Element)
--
-- Exceptions : KEY_ERROR If Process attempts to change the key of any element.
-- The state of the List is examined in Exercise 19.
```

```
private
```

```
-- We'll fill this in later.
```

```
end Sorted_List;
```

## Sample Package Instantiations

This generic list package has the most elaborate set of generic formal parameters that we have used. To instantiate a list we must supply two types and three functions. These five parameters allow us to use the generic list package for many different sorted lists; our list package is very reusable. Let's look at some examples that instantiate list packages from this generic declaration.

First let's examine a list whose elements are inventory records. Each inventory record contains a part name (a string containing no more than 10 characters) and the number in stock. The part name is the key for our inventory record. Using the bounded-length string package developed in Chapter 5 for part names, here are the necessary declarations for our list element:

```
-- Instantiate a string package for part name
package Part_Strings is new Bounded_Strings (Max_Length => 10);

-- Declare a synonym so we won't have to use such a long name
subtype Name_String is Part_Strings.Bounded_String;

-- Inventory record
type Part_Rec is
 record
 Name : Name_String;
 Amount : Natural := 0;
 end record;
```

We now have the types for our list element (`Part_Rec`) and key (`Name_String`). The functions for comparing keys are available from package `Part_Strings`. We must write the function that returns the key of an element. This function simply returns the value of the `Name` field of our part record.

```
function Name_Of (Part : in Part_Rec) return Name_String is
begin
 return Part.Name;
end Name_Of;
```

Here then is the code necessary to instantiate a package for a sorted list of part records:

```
package Part_List is new Sorted_List (Element_Type => Part_Rec,
 Key_Type => Name_String,
 Key_Of => Name_Of,
 "=" => Part_Strings.Equal,
 "<" => Part_Strings."<");
```

Now let's look at an instantiation of a list whose elements are integer values. In a simple list like this one the integer element is the key. We supply `Integer` for both formal generic type parameters `Element_Type` and `Key_Type`. The functions ("`=`" and "`<`") for comparing integer values are predefined by Ada in package `Standard`. That only leaves one more generic parameter: the function that returns the key of an element. Again, we must write that function ourselves. Because the key and element of this simple list of integers are the same, this function just needs to return the element. We call this function `Identity`. Here is the code for it:

```
function Identity (Element : in Integer) return Integer is
begin
 return Element;
end Identity ;
```

Now that we know all of the actual parameter values to supply, we can write the instantiation for our simple list of integers.

```
-- Instantiate a class for a sorted list of Integers.
package Integer_List is new Sorted_List (Element_Type => Integer,
 Key_Type => Integer,
 Key_Of => Identity,
 "=" => Standard."=",
 "<" => Standard."<");
```

### Sample Iterator Instantiations

Once we have instantiated a list package like `Part_List`, we can call `Part_List.Clear`, `Part_List.Insert`, `Part_List.Delete`, `Part_List.Modify`, `Part_List.Retrieve`, `Part_List.Full`, `Part_List.Empty`, and `Part_List.Length`. However, we cannot call `Part_List.Traverse`. `Traverse` is a generic procedure. This generic procedure is used to instantiate any iterator needed by our application. For example, suppose for reordering purposes, we would like to display the name of every part for which there are less than 10 in stock. This requires that we go through our entire list of parts. For each part we check to see if there are less than 10 in stock. If there are less than 10, we display the part name. We write the code for processing one part in a procedure whose parameter profile must match that of the generic formal procedure `Process` associated with our generic procedure `Part_List.Traverse`. Here is the necessary procedure for checking the number in stock of a particular part:

```
procedure Check_One_Part (Part : in out Part_Rec) is
begin
 if Part.Amount < 10 then
 Part_Strings.Put (Item => Part.Name);
 Text_IO.New_Line;
 end if;
end Check_One_Part;
```

Now we are ready to instantiate our iterator that checks the amount in stock of every part in our inventory.

```
procedure Check_Inventory is new
 Part_List.Traverse (Process => Check_One_Part);
```

Procedure `Check_Inventory` is an iterator that we have custom built for our application from the generic iterator `Traverse`. Here is an example of using it to go through all the parts in an auto parts list:

```
-- Declare a list object
Auto_Part_List : Part_List.List_Type (Max_Size => 500);
.
.
.
```

```
-- Display all part names with less than 10 in stock
Check_Inventory (List => Auto_Part_List);
```

Let's look at a second example of instantiating `Traverse`. This time we want to create an instance that displays all of the negative numbers in a list of type `Integer_List.List_Type`. First, we must write a procedure to process *one element* of our list. This procedure checks that the element is negative and prints it if it is.

```
procedure Put_If_Negative (Element : in out Integer) is
begin
 if Element < 0 then
 Integer_IO.Put (Item => Element, Width => 6);
 Text_IO.New_Line;
 end if;
end Put_If_Negative;
```

To create our iterator, `Put_If_Negative` is matched with the generic formal parameter `Process`.

```
procedure Put_All_Negatives is new Traverse (Process => Put_If_Negative);
```

Here is an example of using this iterator to display all of the negative numbers in a list:

```
-- Declare a list object
Number_List : Integer_List.List_Type (Max_Size => 100);
.
.
.
-- Display all the negative numbers in the list
Put_All_Negatives (List => Number_List);
```

We give additional examples of instantiating iterators from `Traverse` in the application developed in the next section. In Chapter 9 we discuss methods for instantiating iterators with additional capabilities from `Traverse`.

### The Application Level

We can use the list specification presented in the last section in a variety of applications that need a collection of ordered elements. In this section we look at one application in detail.

#### *An Electronic Address Book*

You just purchased the latest model portable digital assistant, a pocket-sized computer that helps you keep notes, schedule appointments, and look up important information. This model is so new that the software for the "Address Book" has not been released yet. Because you already have a generic list package available, you decide that you can easily write your own address book program to keep track of all your friends. First you need to write a specification for your program. Let's consider what you should be able to do with

an address book. Of course you should be able to look up a name, add a new name (along with relevant information), and delete a name. We also should be able to change the information when a friend moves. And there is another type of change our program should handle. Our friend Suzanne Pawlan was married last July 3rd and became Suzanne Levy. So our program had better allow us to change a person's name. Let's also make it possible to find out if any of our friends have a birthday coming up this month. Finally, let's make it possible to print out address labels. One more thing: because we want to keep all our data between runs of our program, we need to be able to save our list information on a disk.

Most portable digital assistants allow a pen to be used to choose from menus of commands and even to "write" input data. Because your Ada compiler does not have a package that supports pen input, you decide to use `Text_IO` for the time being. However, you expect to purchase a `Pen_IO` package later this year. Therefore we should write the program so it can easily be changed from using keyboard input to pen input.

Here is a more complete specification of the desired program:

**Specification:**  
**Address Book**

***Function***

This program implements an Address Book.

***Input***

A command specifying an operation to be done. Depending on the command, there may be additional input values (the data). Here are the commands and other data required.

| <i>Command</i> | <i>Data Required</i>                                                                                                                                              |
|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Add            | First Name<br>Last Name<br>Birthday (month, day, and year)<br>Phone Number<br>Street Address<br>City<br>State<br>Zip Code                                         |
| Delete         | First Name<br>Last Name                                                                                                                                           |
| Lookup         | First Name<br>Last Name                                                                                                                                           |
| Change         | First Name<br><br>Last Name<br><br>One or more of the following<br>Birthday (month, day, and year)<br>Phone Number<br>Street Address<br>City<br>State<br>Zip Code |

|             |                                                        |
|-------------|--------------------------------------------------------|
| Change_Name | Old Name (First and Last)<br>New Name (First and Last) |
|-------------|--------------------------------------------------------|

Birthdays

Labels

Quit

### ***Output***

The output depends on the command. In the following table we do not include prompt messages or error messages.

| <i>Command</i> | <i>Output</i>                                                |
|----------------|--------------------------------------------------------------|
| Add            | Confirm addition                                             |
| Delete         | Confirm deletion                                             |
| Lookup         | All data for the person                                      |
| Change         | Confirm change of data                                       |
| Change_Name    | Confirm change of name                                       |
| Birthdays      | Names of people with birthdays this month                    |
| Labels         | Name and mailing address for each person in the address book |
| Quit           |                                                              |

### ***Processing Requirements***

1. The program must supply helpful prompts and meaningful error messages to the user. It should be robust (not crash easily).
2. The user-input part of the program should be implemented to allow it to be replaced at a later date with a different user interface.
3. The data must be stored on disk between program runs.

### ***Assumptions***

1. There are no duplicate names.
2. No first name, last name, street address, or city contains more than 20 characters.
3. Our list of friends will not contain more than 500 people.

### ***The Design***

As always, you start with a design phase. The main data object is our address book, which is some kind of a *list of friends*. Let's call this object `Address_Book`. As we need only one address book for our friends, we can implement this object as an ADO. The operations on `Address_Book` are the commands given in our problem specification.

Each friend has several pieces of data associated with him or her:

1. a name

2. an address
3. a phone number
4. a birthday

This description of a friend suggests a record with four fields: one for each of the four pieces of data. All of these pieces of data have subdata associated with them. Should we make each of these fields a record? Let's look at them one at a time. A name consists of a first name and a last name. A record is a good way to keep track of these two different parts of a name. Likewise a record is a good way to store the four parts (street, city, state, and zip) that make up an address. What about the phone number? We could store it as a single 10-digit number or as a record with a 3-digit area code and 7-digit number. Because it is possible in the future that we may want to process friends by area code, let's use a record to separate the area code from the number. A birthday seems like an excellent candidate for using a record; there are three fields—day, month, and year. But there is an alternative. Instead of defining our own record to store a date, we can use the types in package `Calendar`. This package contains several useful operations for dealing with dates that make it easier for us to write the program. Here is a declaration of a record type that reflect these choices:

```
type Friend_Rec is
 record
 Name : Name_Rec;
 Address : Address_Rec;
 Phone : Phone_Rec;
 Birthday : Calendar.Time;
 end record;
```

Much of the data in the nested record types are strings. As the assumptions in the specification state that none of the strings contains over 20 characters, we can use the bounded string class we developed in Chapter 5. Some data, such as state abbreviations, contain a fixed number of characters. Ada's predefined fixed length string type `String` is suitable for these fields. The complete details of these nested record types are given in Specification 8-2.

### Specification 8-2— Address Book ADO

```
with Bounded_Strings;
with Calendar;
package Address_Book is

 -- This abstract data object models an address book.

 -- Bounded String Class
 package String20 is new Bounded_Strings (Max => 20);

 -- Bounded String Types
 subtype Name_String is String20.Bounded_String;
 subtype Address_String is String20.Bounded_String;
```

```

-- Fixed Length String Types
subtype State_String is String (1..2);
subtype Zip_String is String (1..5);
subtype Area_Code_String is String (1..3);
subtype Phone_Num_String is String (1..7);

type Name_Rec is
 record
 First : Name_String;
 Last : Name_String;
 end record;

type Address_Rec is
 record
 Street : Address_String;
 City : Address_String;
 State : State_String;
 Zip : Zip_String;
 end record;

type Phone_Rec is
 record
 Area : Area_Code_String;
 Number : Phone_Num_String;
 end record;

type Friend_Rec is
 record
 Name : Name_Rec;
 Address : Address_Rec;
 Phone : Phone_Rec;
 Birthday : Calendar.Time;
 end record;

-- Exceptions
DUPLICATE_NAME : Exception;
MISSING_NAME : Exception;

procedure Add (Friend : in Friend_Rec);
-- This procedure adds a friend to the address book.

-- Preconditions : None
--
-- Postconditions : Address Book = Original Address Book + Friend
--
-- Exceptions : DUPLICATE_NAME If the name of Friend is already in the address
-- book, the address book is not changed.

procedure Delete (Name : in Name_Rec);
-- This procedure deletes a friend from the address book.
-- Name identifies the friend to delete.

-- Preconditions : None

```



```

--
-- Postconditions : Address Book = Original Address Book - friend
--
-- Exceptions : MISSING_NAME If the Name is not found in the
-- address book, the address book is
-- not changed.
--

procedure Lookup (Name : in Name_Rec;
 Friend : out Friend_Rec);
-- This procedure looks up the Name in the address book and
-- returns the information associated with that name.
--
-- Preconditions : None
--
-- Postconditions : Friend = Information associated with Name.
-- The address book is not changed.
--
-- Exceptions : MISSING_NAME If the Name is not found in the address book,
-- the address book is not changed.
--

procedure Change (Friend : in Friend_Rec);
-- This procedure replaces information on a friend with new information.
--
-- Preconditions : None
--
-- Postconditions : The information in Friend replaces the information for the friend
-- in the address book.
--
-- Exceptions : MISSING_NAME If the name given in Friend is not found in the
-- address book, the address book is not changed.
--

procedure Change_Name (Old_Name : in Name_Rec;
 New_Name : in Name_Rec);
-- This procedure changes a friend's name.
--
-- Preconditions : None
--
-- Postconditions : The New_Name replaces Old_Name in the address book.
--
-- Exceptions : MISSING_NAME If Old_Name is not found in the address
-- book, the address book is not changed.
-- DUPLICATE_NAME If New_Name is already in the address
-- book, the address book is not changed.
--

procedure Birthdays;
-- This procedure displays in alphabetical order (to the standard output file)
-- the names of all friends having a birthday this month.

```

```

-- Preconditions : None
--
-- Postconditions : The names of all friends with birthdays this month are displayed.

procedure Labels;
-- This procedure prints (to the standard output file) mailing
-- labels for all friends in the address book.

-- Preconditions : None
--
-- Postconditions : Mailing labels are printed in alphabetical order for all friends.

procedure Save;
-- This procedure saves the current address book to disk.
-- This procedure should be called before exiting the program.

-- Preconditions : None
--
-- Postconditions : The current address book is saved for the next program run.

end Address_Book;

```

### ***The Implementation***

You probably have noted the similarity between this address book object and sorted list class we developed earlier this chapter. The address book is a kind of sorted list. Recall from Chapter 1 that the *kind of* hierarchy is useful for classifying things. This hierarchy of abstractions is also useful for determining how software components can be reused in different applications. In this application, we can use our list class as the foundation for our address book object; we implement the address book operations with list operations. The relationship between the address book ADO and the sorted list ADT is shown in Figure 8-1. Our address book ADO body instantiates a sorted list package whose elements are type `Friend_Rec` and whose keys are type `Name_Rec`. We need to include functions that return the name portion of a `Friend_Rec` and that compare names as actual generic subprogram parameters.

One important feature of the address book is that it must be stored on disk between runs. When our application program starts, the *retained data* on the file automatically is brought into the address book. The retained data is the same information contained in the address book but stored in some kind of file format. Figure 8-1 illustrates the relationship between the retained data and our address book. What sort of file should we use for this data? One approach is to keep the data in a text file, reading and writing all the record fields one at a time. Using a text file requires you to convert back and forth between the way data is formatted in the text file and the way the data is stored in a record. A simpler alternative is to use a *binary* file. Package `Sequential_IO`, whose specification is given in Appendix E, allows us to read and write records without these conversions.

The save operation for the address book ADO can be implemented by instantiating a list iterator that writes a copy of each list element to the file. But how can we automatically

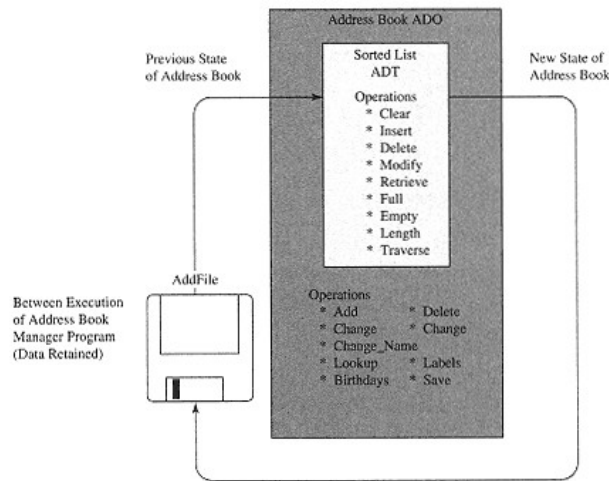


Figure 8-1  
The Program Data (a Sorted List) and Retained Data (AddFile)

bring this data into our program when the program starts? To solve this problem, we make use of a feature of Ada that you may not have seen before. Here is the EBNF definition of a package body:

```
package_body ::= package body package_simple_name is
 [declarative_part]

 [begin
 sequence_of_statements]

 end [package_simple_name];
```

Note the optional sequence of statements at the end of a package body. Until now, our package bodies consisted only of the declarative part. Here we declared local types, constants, variables, and subprograms for our package body as well as the bodies for the subprograms declared in the package declaration. The statements in the optional sequence are executed when the package body is elaborated. Because the package body is elaborated before any of its subprograms are called, we can use this code to initialize an abstract data object. For our address book, this initialization consists of reading the retained data from the disk. We are now ready to write the body of our address book ADO. As you go through Body 8-1, pay particular attention to the declaration of the local procedures and how they are used as actual generic formal parameters in the instantiation of the list package and the three iterators required for the ADO.

**Body 8-1—****Address Book Implements Specification 8-2**

```

with Sequential_IO;
with Sorted_List;
with Text_IO;

package body Address_Book is

 -- Package for saving address book between program runs
 package Friend_IO is new Sequential_IO (Element_Type => Friend_Rec);

 -- Declare functions needed for instantiating a sorted list package

 function Name_Of (Friend : in Friend_Rec) return Name_Rec;
 function Equal (Left : in Name_Rec; Right : in Name_Rec) return Boolean;
 function "<" (Left : in Name_Rec; Right : in Name_Rec) return Boolean;

 package Sorted_Friend_List is new Sorted_List (Element_Type => Friend_Rec,
 Key_Type => Name_Rec,
 Key_Of => Name_Of,
 "=" => Equal,
 "<" => "<");

 -- Declare procedures needed for instantiating iterators

 procedure Check_One_Birthday (Friend : in out Friend_Rec);
 procedure Print_One_Label (Friend : in out Friend_Rec);
 procedure Write_One_Friend (Friend : in out Friend_Rec);

 -- Objects global to the package body

 -- The Address Book Object
 The_Book : Sorted_Friend_List.List_Type (Max_Size => 500);

 -- Sequential file for saving the address book between runs
 File_Name : constant String := "Friends.Bin";
 Friend_File : Friend_IO.File_Type;
 Friend : Friend_Rec; -- One file element

 -- Instantiate iterator procedures

 procedure Print_Birthdays is new
 Sorted_Friend_List.Traverse (Process => Check_One_Birthday);
 procedure Print_Labels is new
 Sorted_Friend_List.Traverse (Process => Print_One_Label);
 procedure Write_Book is new
 Sorted_Friend_List.Traverse (Process => Write_One_Friend);

```

```
-- Local Subprogram Bodies
```

```

function Name_Of (Friend : in Friend_Rec) return Name_Rec is
begin
 return Friend.Name;
end Name_Of;
```

```

function Equal (Left : in Name_Rec; Right : in Name_Rec) return Boolean is
begin
 return String20.Equal (Left.First, Right.First) and
 String20.Equal (Left.Last, Right.Last);
end Equal;
```

```

function "<" (Left : in Name_Rec; Right : in Name_Rec) return Boolean is
 use String20;
begin
 return Left.Last < Right.Last
 or
 (String20.Equal (Left.Last, Right.Last) and
 Left.First < Right.First);
end "<";
```

```

procedure Check_One_Birthday (Friend : in out Friend_Rec) is
begin
 -- Compare the friend's birthday month to the current month
 if Calendar.Month(Friend.Birthday) = Calendar.Month(Calendar.Clock) then
 String20.Put (File => Text_IO.Standard_Output,
 Item => Friend.Name.First);
 Text_IO.Put (' ');
 String20.Put (File => Text_IO.Standard_Output,
 Item => Friend.Name.Last);
 Text_IO.New_Line;
 end if;
end Check_One_Birthday;
```

```

procedure Print_One_Label (Friend : in out Friend_Rec) is
begin
 -- The name
 String20.Put (File => Text_IO.Standard_Output,
 Item => Friend.Name.First);
 Text_IO.Put (' ');
 String20.Put (File => Text_IO.Standard_Output,
 Item => Friend.Name.Last);
 Text_IO.New_Line;
 -- The street address
 String20.Put (File => Text_IO.Standard_Output,
 Item => Friend.Address.Street);
 Text_IO.New_Line;
 -- City, State, and Zip
 String20.Put (File => Text_IO.Standard_Output,
 Item => Friend.Address.City);
```

```

Text_IO.Put ("");
Text_IO.Put (Friend.Address.State);
Text_IO.Put (' ');
Text_IO.Put (Friend.Address.Zip);
Text_IO.New_Line (3); -- Spacing to next label
end Print_One_Label;

```

```

procedure Write_One_Friend (Friend : in out Friend_Rec) is
begin
 Friend_IO.Write (File => Friend_File, Item => Friend);
end Write_One_Friend;

```

```

-- Operation Bodies

```

```

procedure Add (Friend : in Friend_Rec) is
begin
 Sorted_Friend_List.Insert (List => The_Book, Item => Friend);
exception
 when Sorted_Friend_List.DUPLICATE_KEY =>
 raise DUPLICATE_NAME; -- Convert list exception to address book exception
end Add;

```

```

procedure Delete (Name : in Name_Rec) is
begin
 Sorted_Friend_List.Delete (List => The_Book, Key => Name);
exception
 when Sorted_Friend_List.KEY_ERROR =>
 raise MISSING_NAME; -- Convert list exception to address book exception
end Delete;

```

```

procedure Lookup (Name : in Name_Rec;
 Friend : out Friend_Rec) is
begin
 Sorted_Friend_List.Retrieve (List => The_Book,
 Key => Name,
 Element => Friend);
exception
 when Sorted_Friend_List.KEY_ERROR =>
 raise MISSING_NAME; -- Convert list exception to address book exception
end Lookup;

```

```

procedure Change (Friend : in Friend_Rec) is
begin
 Sorted_Friend_List.Modify (List => The_Book,
 Element => Friend);
exception
 when Sorted_Friend_List.KEY_ERROR =>
 raise MISSING_NAME; -- Convert list exception to address book exception
end Change;

```

```

procedure Change_Name (Old_Name : in Name_Rec;
 New_Name : in Name_Rec) is
 Friend : Friend_Rec;
begin
 -- Retrieve the old information
 Sorted_Friend_List.Retrieve (List => The_Book,
 Key => Old_Name,
 Element => Friend);

 -- Change the name
 Friend.Name := New_Name;
 -- Add the new name to the address book
 Sorted_Friend_List.Insert (List => The_Book, Item => Friend);
 -- Delete the old name from the address book
 Sorted_Friend_List.Delete (List => The_Book, Key => Old_Name);
exception
 when Sorted_Friend_List.KEY_ERROR =>
 raise MISSING_NAME; -- Convert list exception to address book exception
 when Sorted_Friend_List.DUPLICATE_KEY =>
 raise DUPLICATE_NAME; -- Convert list exception to address book exception
end Change_Name;

procedure Birthdays is
begin
 Print_Birthdays (List => The_Book); -- Call iterator instantiated above
end Birthdays;

procedure Labels is
begin
 Print_Labels (List => The_Book); -- Call iterator instantiated above
end Labels;

procedure Save is
begin
 Friend_IO.Create (File => Friend_File, Name => File_Name);
 Write_Book (List => The_Book); -- Call iterator instantiated above
 Friend_IO.Close (Friend_File);
end Save;

begin -- Initialize the Address Book

Exception_Block: -- To handle creation of nonexistent file
begin
 Friend_IO.Open (File => Friend_File,
 Mode => Friend_IO.In_File,
 Name => File_Name);
 loop -- Each iteration, one friend is read from the file and added to the book
 exit when Friend_IO.End_Of_File (Friend_File);
 Friend_IO.Read (File => Friend_File, Item => Friend);
 Sorted_Friend_List.Insert (List => The_Book, Item => Friend);
 end loop;

```

```

 Friend_IO.Close (Friend_File);

exception -- Handles first run of program where there is no saved address book
 when Friend_IO.NAME_ERROR =>
 Text_IO.Put_Line ("Creating the initial Address Book");
 end Exception_Block;

end Address_Book;

```

We have included an exception handler for our initialization code. The binary file with our saved data does not exist the first time our program is run. Attempting to open a nonexistent file will raise a `NAME_ERROR` exception. The exception handler simply displays a message informing the user that it is creating the initial address book. Because the list object is initialized to empty when `The_Book` is elaborated, no other action is necessary in our exception handler.

Now we need only write the program that uses the address book ADO to implement our specification. This program obtains commands and data from the user and uses the operations in our address book ADO to carry them out. We leave the implementation of a `Text_IO` based version of this program as an exercise.

In our address book ADO we have made calls to the list operations specified for the sorted list ADT, creating and modifying lists *without even knowing how the list is implemented*. At a user level, these are logical operations on a list. At a lower level, these operations are implemented as Ada procedures or functions that manipulate an array or other data-storing medium holding the list's elements. As we have seen with stacks and queues, there are multiple functionally correct ways to implement an abstract data type. Between the user picture and the eventual representation in the computer's memory, there are intermediate levels of abstractions and design decisions. For instance, how will the logical order of the list elements be reflected in their physical ordering? Will the list be represented as a sequential or a linked structure? If a linked structure is chosen, will its elements be stored in statically or dynamically allocated memory?

### The Implementation Level

We said earlier that the logical order of the list elements may or may not be mirrored in the way that they are physically stored in a data structure. However, the way that the list elements are arranged physically affects the way that we access the elements in the list. This arrangement may have implications on the efficiency of the list operations. For instance, there is nothing in the specification of the sorted list ADT that requires us to implement the list with the elements *stored* in order. If we stored the elements in an array, completely unordered, we still could implement all the list operations. With this list representation, the insert algorithm could be an  $O(1)$  operation, because we do not care about the order of the elements. (If the elements were ordered, the insert operation would be  $O(N)$ , as we shall discuss later.) However, the traverse operation, which requires that the elements be processed *in order*, would be  $O(N^2)$  (or at best,  $O(N \log_2 N)$ , as we see in Chapter 14). For a list whose elements are physically sorted, this operation is  $O(N)$ . We leave further exploration of the implementation of the sorted list ADT in an unordered structure as an exercise.



In this section we develop two list representations that preserve the order of the list elements. That is, the elements are physically stored in such a way that, from one list element, we can access its logical successor directly. We first look at a sequential list representation. The distinguishing feature of a sequential implementation is that the elements are stored sequentially, in adjacent slots in an array. The order of the elements is implicit in their placement in the array.

Array-based list implementations are so common that many people refer to an "array" of data when they really mean a "list," as if the two were interchangeable. Of course, we know this to be a misnomer—an array is one place to store list elements, but it is not the only implementation choice.

The second approach that we discuss is a *linked list* representation. In a linked implementation, the data elements are not constrained to be stored in physically contiguous, sequential order; rather the individual elements are stored somewhere in memory, and their order is maintained by explicit links between them.

We want to emphasize that the choice between sequential and linked list representations is not the same as the choice between static and dynamic storage allocation. These are separate issues. We typically store arrays in variables that have been declared, as illustrated in Figure 8-2(a), but an array-based implementation does not necessarily use static storage. The whole array could exist in a dynamically allocated area of memory; that is, we could get space for the whole structure at once using the new allocator, as illustrated in Figure 8-2(b).

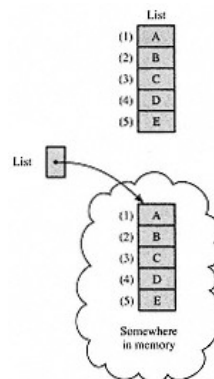


Figure 8-2  
Array Based Lists in Static and Dynamic Storage

We tend to think of linked structures as being in dynamically allocated storage, as illustrated in Figure 8-3(a), but we can implement linked lists without dynamic allocation. A linked list can be implemented in an array; the elements might be stored in the array in any order, and "linked" by their indexes [Figure 8-3(b)]. We discuss this approach in detail in Chapter 10.

We develop both sequential and linked implementations of a list of elements. In each of these implementations we use the generic formal function `Key_Of` to obtain the key of an element. Keys then are compared with the generic formal functions `"="` and `"<"`. All three of these functions are supplied by the user of our generic list package when they instantiate it.

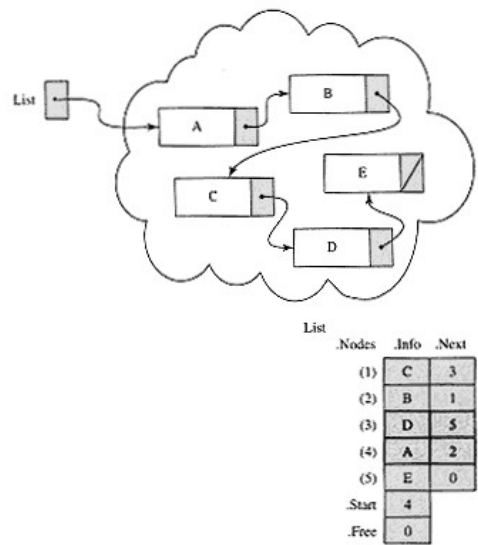


Figure 8-3  
Linked Lists in Static and Dynamic Storage

### Sequential List Implementations

The elements in the sequential list implementation are stored in an array:

```
type Element_Array is array (Positive range <>) of Element_Type;

Items : Element_Array (1..Max_Size);
```

What makes this a sequential representation is not the array itself but the *arrangement* of the elements within the array. The first (smallest key value) element is stored in the first array slot, the second (next smallest key value) element in the second slot, and so on. Now we know where the list begins—in the first array slot. Where does the list end? The *array* ends at the slot with index `Max_Size`, but the *list* may not fill the whole array. Therefore, we keep track of the number of list elements; we call this value *Length*. Now we know that

the last (largest key value) element in the list is in the array slot with index `Length`. As we did for stacks and queues, we bundle together the array and its supporting data in a discriminated record type. Here are the necessary declarations for the private part of Specification 8-1 (page 309):

```
private

type Element_Array is array (Positive range <>) of Element_Type;

type List_Type (Max_Size : Positive) is
 record
 Length : Natural := 0;
 Items : Element_Array (1..Max_Size);
 end record;

end Sorted_List;
```

Figure 8-4 depicts a sequential list containing five elements. The elements are ordered by their keys. For simplicity, only the key values (names in this example) are shown in the figure. The element whose key comes first alphabetically is in `List.Items (1)`, the next element is in `List.Items (2)`, and so on. The element whose key comes last alphabetically is in `List.Items (List.Length)`.

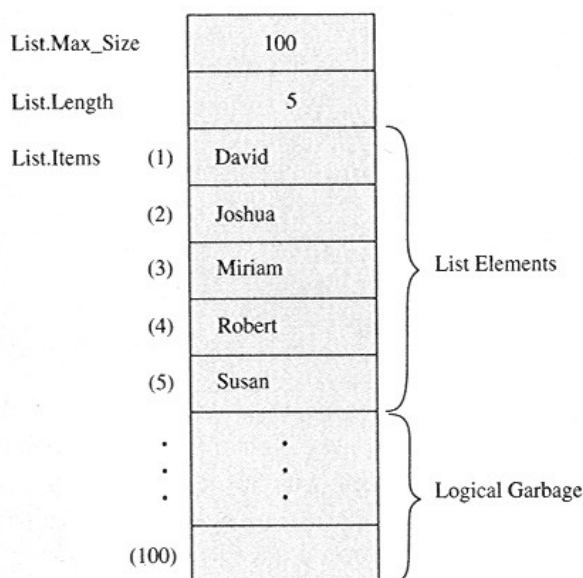


Figure 8-4  
A Sequential List

Now let's look at the operations we have specified for the List class. Because of the structural resemblance of this implementation to the array-based stack, several of the operations may seem familiar to you. An empty list is one whose `Length` field is zero. Thus procedure `Clear` and function `Empty` merely set or check the value of the `Length` field. They can be coded without additional decomposition.

Like the array-based implementations of stacks and queues, this list implementation requires a full operation, to make sure that we do not try to insert another element when all the array slots are full. Because the array has `Max_Size` slots, the full operation simply compares the list's `Length` to the record discriminant `Max_Size`.

## Finding a List Element

The list operations to retrieve, modify, insert, and delete an element require searching the list to find the element to process or to find the appropriate place to insert. To support these operations, let's write a procedure `Search_Array`. This procedure takes a *slice* of the list's array field and a key value as inputs and searches the array for the element with this key. The array slice we pass to `Search_Array` is the portion of the list array containing list elements; the logical garbage is not passed (Figure 8-4).

If the element is found, `Search_Array` returns the index of the desired element's location (a value between 1 and `Max_Size`). What if the element with the specified key is not found, as in the case when we are inserting a new element? If the element is not found, the procedure could return a value of zero for the location. However, as long as we're searching the array, it would be useful to know where an element with this key would belong if it *were* in the array. Therefore, `Search_Array` also returns the index of the array where this key would be inserted. Specification 8-3 defines this search procedure.

### Specification 8-3— Array Search

```
procedure Search_Array (The_Array : in Element_Array;
 Key : in Key_Type;
 Found : out Boolean;
 Location : out Positive);

-- Searches for the location of Key in The_Array

-- Preconditions : The_Array is sorted in ascending order
--
-- Postconditions : Found is True and Key = Key_Of (The_Array (Location))
-- or
-- Found is False and Key > keys of The_Array (1..Location_1) and
-- Key < keys of The_Array (Location..The_Array'Last)
```

For simplicity we use a *sequential search*. We use the generic formal function `Key_Of` to obtain the key of a list element. We start at the beginning of the list and loop until (1) we have found an element with the desired key, (2) we have passed all the elements in the list whose keys are smaller than `Key`, or (3) we have reached the end of the list. Because we pass this search procedure a slice of the list array without the logical garbage (Figure 8-4), the last item in the list is at position `The_Array'Last`. We set the output parameters based on the loop terminating conditions. Using these exit conditions, we can derive the following loop structure.

```
Index := The_Array'First; -- Start at the beginning of the array
loop -- Each iteration, one array element is checked
 exit when Key = Key_Of (The_Array(Index)) or -- Found it
 Key < Key_Of (The_Array(Index)) or -- Passed by
 Index > The_Array'Last; -- Reached the end
 Index := Index + 1; -- Go on to next element
end loop;
```

A problem occurs with this loop structure when `Key` is larger than any of the keys in the array. In this case, `Index` will be greater than `The_Array'Last` and attempting to use it as the array index raises `CONSTRAINT_ERROR`. Rearranging the order of the conditions in the exit statement is not the complete solution. `CONSTRAINT_ERROR` is still be raised in the following exit statement:

```
exit when Index > The_Array'Last or
 Key < Key_Of (The_Array(Index)) or
 Key = Key_Of (The_Array(Index));
```

When using the `and` or `or` operators, *all* of the terms in a Boolean expression are evaluated. Thus even when the first term in the above exit statement evaluates to True, the remaining two terms still are evaluated; `CONSTRAINT_ERROR` is raised. Ada has other logical operators, called *short-circuit operators*, that stop the evaluation of additional terms in a Boolean expression as soon as the result is known. The two short-circuit operators are `and then` and `or else`.

How can we determine the result of a Boolean expression without evaluating all of the terms? The `and` operator requires both operands to be True in order for the overall result to be True. If either or both of the operands are False, `and` makes the entire result False. Thus, if the first operand is False, we don't even need to look at the second term. We already know that the expression is False. If either of the operands with an `or` operator is True, then the expression is True. So if the first operand is True, we don't need to look at the second operand. This logic is exactly what the `and then` and `or else` operators perform. The following tables summarize these two logical operators. The question marks for Y are used to indicate that we have not yet looked at Y's value.

| <i>X</i> | <i>Y</i> | <i>X and then Y</i> |
|----------|----------|---------------------|
| False    | ????     | False               |
| True     | False    | False               |
| True     | True     | True                |

| <i>X</i> | <i>Y</i> | <i>X or else Y</i> |
|----------|----------|--------------------|
| True     | ????     | True               |
| False    | True     | True               |
| False    | False    | False              |

Here is our exit statement written with short-circuit operators. When `Index` becomes greater than `The_Array'Last`, the remaining terms in the Boolean expression are not evaluated; `CONSTRAINT_ERROR` is not raised.

```
exit when Index > The_Array'Last or else
 Key < Key_Of (The_Array(Index)) or else
 Key = Key_Of (The_Array(Index));
```

## Using Loop Invariants

We can use loop invariants to develop the loop conditions for our sequential search loop. There are three conditions mentioned on page 330 that cause the search to stop. The first two conditions stop the search when we have passed the point in the array where our key value might be found. Because the array is ordered by key value, searching the entire array isn't necessary; we only need to search until we find a key that is greater than or equal to the one we are looking for. This tells us that on entrance to the loop, all the key values that have already been examined are less than `Key`.

*Loop Invariant Part 1:* All key values in `The_Array(1..Index - 1)` are less than `Key`.

The third condition that stops the search is that there are no more elements left to examine; that is, we have gotten to the end of the array. (The condition occurs when `Key` is larger than any key in the array.) We now know the range of values that `Index` might have.

*Loop Invariant Part 2:* `The_Array'First <= Index <= The_Array'Last + 1`

Let's review what the loop invariant tells us: Part 1 says not to search after passing the place in the array where `Key` might be; part 2 says not to search after the last array element has been examined. Each part of the loop invariant contributes to the exit condition. The resulting exit statement is

```
exit when Index > The_Array'Last or else -- (Part 2)
 Key_Of (The_Array(Index)) > Key; -- (Part 1)
```

When the loop terminates, the loop invariant is still true, and the loop condition has just turned true. Are we sure that the loop terminates? Looking at the inside of the loop we see that on every iteration, `Index` is incremented. Eventually, if we don't pass the location of the correct element, `Index` is incremented past `The_Array'Last`. So we know that the loop eventually terminates.

The only problem with coding the exit statement with these two conditions is that the `>` operator for comparing keys is not one of the two supplied as generic formal parameters. However, we can rearrange the logic of the condition in this exit statement so that we can instead use the `"<"` and `"="` functions supplied as generic parameters. Here is the resulting statement:

```
exit when Index > The_Array'Last or else -- (Part 2)
 Key < Key_Of (The_Array(Index)) or else -- (Part 1)
 Key = Key_Of (The_Array(Index)); -- (Part 1)
```

After exiting this loop, we must assign values to our two `out` parameters `Found` and `Location`. There are three possible situations:

1. `Index > The_Array'Last`. If we reach the end of the array without finding an element whose key is as large as `Key`, then the element is not in the array. `Found` should be set to `False`. Because `Key` is larger than any in the array, it is inserted after

the last element. Therefore we set `Location` to `The_Array'Last + 1`, which is exactly the value that `Index` has when the loop terminates. [See Figure 8-5(a).]

2. `Index <= The_Array'Last` and `Key < Key_Of (The_Array(Index))`.. In this case, we have passed the location where the element belongs, but it isn't in the list. If we wanted to insert an element with `Key`, it would belong in the slot indicated by `Index`. Therefore, `Location` is set to `Index`. [See Figure 8-5(b).]

3. `Index <= The_Array'Last` and `Key = Key_Of (The_Array(Index))`.. In this case, we have found the element within the list. `Location` is set to `Index`. [See Figure 8-5(c).]

The following code assigns values to `Location` and `Found` that satisfy all three cases. We again make use of a short-circuit Boolean operator to prevent `CONSTRAINT_ERROR` from being raised when `Index <= The_Array'Last`.

```
Location := Index;
if Index <= The_Array'Last and then Key = Key_Of (The_Array(Index)) then
 Found := True;
else
 Found := False;
end if;
```

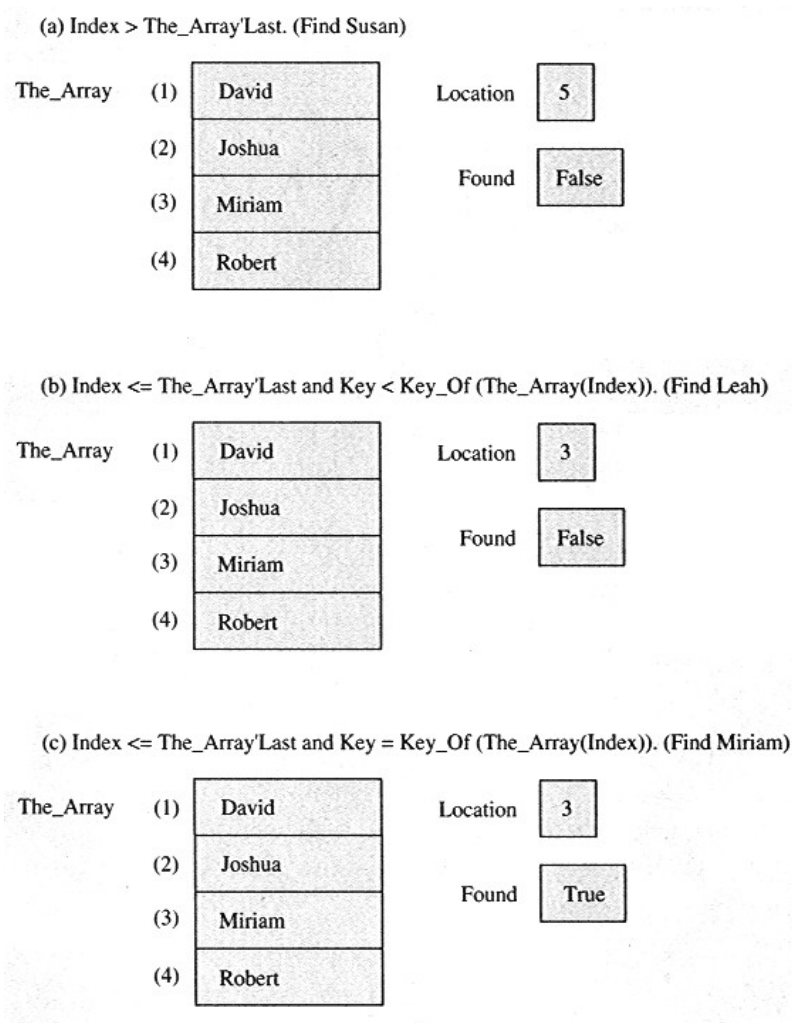


Figure 8-5  
Sequential Search of an Array

Notice that when the Boolean expression in this if statement evaluates to True, we assign True to Found. When it evaluates to False, we assign False to Found. We can obtain the same results using a single assignment statement that reduces the six lines to

```
Location := Index;
Found := Index <= The_Array'Last and then
 Key = Key_Of (The_Array(Index));
```

Note that procedure Search\_Array is not included in our sorted list package declaration (Specification 8-1). Should we add it? No—this procedure is provided for the use of the list operations, not for the application programmer using the package. It is logically *internal* to the sorted list class. If we do not want the list user to know anything about how the list is stored, there is no reason to provide information about the inner workings (array indexes, pointers, or whatever) of the list implementation. Our goal is information hiding; therefore, this procedure is off-limits to the application programmer.

**The Retrieve and Modify Operations**

The Retrieve operation allows the list user to obtain a copy of the list element with a specified key, if that element exists in the list. To do so, we simply search the list's array with Search\_Array. If the search is successful, we return a copy of the element whose array index is given by Location. If the search is not successful, we raise the KEY\_ERROR exception. The Retrieve operation is illustrated in Figure 8-6.

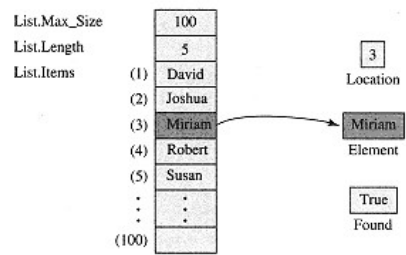


Figure 8-6  
The Retrieve Operation

The Modify operation allows the user to replace an element with another with the same key. As with Retrieve, we use Search\_Array to find the Location of the element to change. Then we copy the replacement element into this array slot. If the search is not successful, we raise the KEY\_ERROR exception. The Modify operation is illustrated in Figure 8-7.

**The Insert Operation**

To add an element to a value-ordered list, we first must find the place where the new element belongs. Unlike a stack (where we always add to the top) or a queue (where we always add to the rear), the list's insertion location is not fixed. Where the element belongs depends on the value of its key field.



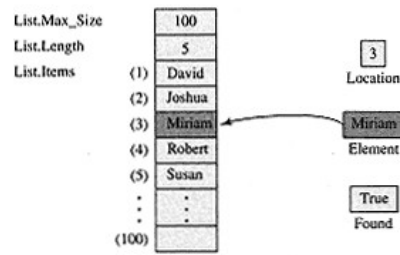


Figure 8-7  
The Modify Operation

We'll use an example to illustrate the insertion operation. Let's say we wish to add an element for Leah to our list. To add it to the sequential list pictured in Figure 8-8 (a), maintaining the alphabetic ordering, we use the following algorithm:

```
Insert

Find the place where the new element belongs
Create space for the new element
Put the new element into the list
```

The first task involves traversing the list until we find the first element that is greater than the one we want to add (in this case, Miriam). `Search_Array` does this task for us; the loop terminates when the insertion point is reached and `Location` is the place for the new element.

Now that we know where the element belongs, we need to create space for it. Because the list is sequential, the element Leah must be put into the slot between its alphabetic predecessor (at index `Location-1`) and successor (at index `Location`). But this slot is occupied. To "create space for the new element," we must move down all the list elements that follow it, from `Location` to `List.Length`. We can move these elements with a loop or with a single assignment statement using array slices. We prefer the assignment method because array slice assignment usually executes much faster than a loop.\* Then we increment the `Length` field of the list.

**Create space for the new element**

```
List.Items (Location + 1 .. List.Length + 1) := List.Items (Location .. List.Length);
List.Length := List.Length + 1;
```

Now the array slot at index `Location` is free; we put the new element in the list in this position. Figure 8-8(b) shows the resulting list.

What if `Search_Array` returns `True` for `Found`? This result means that there is already an element in our list with the same key as the one we are trying to insert. Our specification requires us to raise `DUPLICATE_KEY` and leave the list unchanged.

Does this procedure work if the new element belongs at the beginning or end of the list? Draw yourself a picture to see how the procedure works in each of these cases.

---

\*Most Ada compilers translate array or array slice assignment into a single machine language instruction that moves a block of memory.

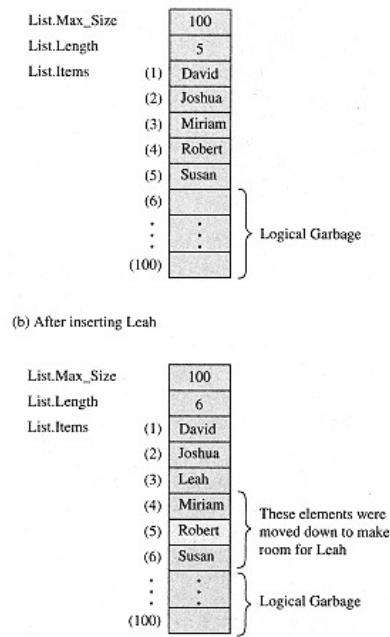


Figure 8-8  
Inserting into a Sequential List

### The Delete Operation

The Delete operation takes two inputs: the list and the key of the element that is to be deleted. We use the following algorithm:

#### Delete Element

```
Find the element in the list
Remove the element from the list
```

The first part of the algorithm is easy—we can call procedure `Search_Array`. If, on return from `Search_Array`, `Found` is `True`, `Location` contains the index of the element with the key we wish to remove from the list. If `Found` is returned as `False`, there is no element in the list with the given key. Our specification requires us to raise `KEY_ERROR`.

How do we "remove the element from the list"? Let's look at our list example [Figure 8-9(a)]. Removing Susan from the list is easy, for hers is the last element in the list [see Figure 8-9(b)]. If we then remove David from the list, we need to move up all the elements that follow to fill in the space [see Figure 8-9(c)]. In the sequential implementation, we do not actually remove the element; instead we cover it up with the element that previously followed it. Each successive element must be moved to cover up its predecessor. As we did

in the insert operation, we can use an assignment statement with array slices to move a group of elements. Finally, we decrement the length of the list.

**Remove the element from the list**

```
List.Items (Location..List.Length-1) := List.Items (Location + 1 .. List.Length);
List.Length := List.Length - 1;
```

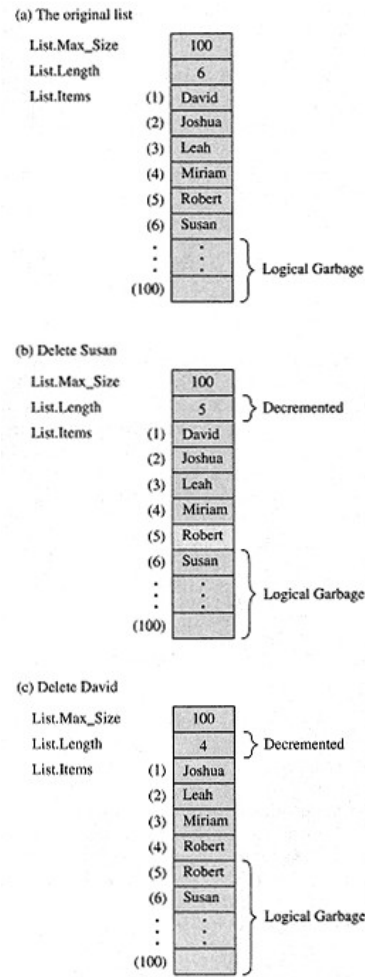


Figure 8-9  
Deleting from a Sequential List

What happens when we are trying to remove the last element in the list, that is, the element at `List.Items(List.Length)`? There are no elements following it in the list, so the value in this slot doesn't get "covered up." Because in this situation `Location + 1` is greater than `List.Length`, the range `Location + 1 .. List.Length` is a null range. `Location .. List.Length - 1` is also a null range. An array slice with a null range contains no elements. Therefore no elements are copied by our assignment statement. Decrementing `List.Length`, in effect, implements the deletion of the element from the list. The deleted element's value is still in the array, but it is no longer in the list. For instance, Susan is still in array slot 6 in Figure 8-9(b), but there are only five elements in the list. This element is logically (though not physically) inaccessible.

### The Traverse Operation

Procedure `Traverse` is a generic procedure that can be instantiated to construct nearly any iterator that the user may need. As we saw in the examples on page 313, the user supplies an actual procedure for the formal generic procedure `Process`. Procedure `Traverse` calls `Process` for every element in the list. Because we know how many elements are in the list (`List.Length`), we can use a for loop to traverse the array of elements in our list record.

#### Traverse

```
for Index in 1..List.Length loop
 Process (Element=>List.Items(Index));
end loop;
```

To allow an application programmer to modify elements in the list, the parameter `Element` for procedure `Process` has a mode of `in out`. There is a danger here. An application programmer could change the key of the element. Making this change would destroy the order of the list. To prevent someone from changing the key while still allowing modification of the remainder of the element, we can pass a *copy* of each element rather than the element itself. When we get the element back from the user's procedure, we can check to see if it has changed the key. If it has, we raise `KEY_ERROR`. Here is the algorithm that includes this check:

#### Traverse

```
for Index in 1..List.Length loop
 Copy := List.Items(Index); -- Make of copy of the element
 Process (Element => Copy); -- Let the user process the copy
 if Key_Of (Copy) /= Key_Of (List.Items(Index)) then
 raise KEY_ERROR; -- User process changed the key
 else
 List.Items(Index) := Copy; -- Put back the possibly modified copy
 end if;
end loop;
```

### Notes on the Sequential List Implementation

In many of the list operations, we declared a local variable `Location`, which contains the array index of the list element being processed. The values of array indexes never get outside of the set of list operations; this information is *internal* to the implementation of the `List` class. If the application programmer wants a record in the list, procedure `Retrieve` does not give them the index of the record; instead it returns a copy of the record itself. If the application programmer

changes values in this record, those changes are not reflected in the list unless they call `Modify`. The list user is never allowed to see or manipulate the physical structure in which the list is stored. These details of the list implementation are hidden; the list class is encapsulated by the ADT.

### Sequential List Package Body

Body 8-2 contains the complete body for our sequential array-based implementation of a sorted list.

#### Body 8-2—

#### Sequential Array-Based List Implements Specification 8-1

```
package body Sorted_List is

 -- Local procedures

 procedure Search_Array (The_Array : in Element_Array;
 Key : in Key_Type;
 Found : out Boolean;
 Location : out Positive) is
 -- Searches for the location of Key in the The_Array
 --
 -- List use note: The_Array is a slice of the list array that contains
 -- all the list elements and no logical garbage.

 -- Preconditions : The_Array is sorted in ascending order.
 --
 -- Postconditions : Found is True and Key = Key_Of (The_Array (Location))
 -- or
 -- Found is False and
 -- Key > Keys of The_Array(1..Location-1) and
 -- Key < Keys of The_Array(Location..The_Array'Last)

 Index : Positive;
 begin
 Index := The_Array'First;
 loop -- Each iteration, one array element is checked
 exit when Index > The_Array'Last or else
 Key < Key_Of (The_Array(Index)) or else
 Key = Key_Of (The_Array(Index));
 Index := Index + 1;
 end loop;

 -- Assign results to out parameters
 Found := Index <= The_Array'Last and then Key_Of (The_Array(Index));
 Location := Index;
 end Search_Array;
```

```
-- Operations
```

```

procedure Clear (List : in out List_Type) is
begin
 List.Length := 0;
end Clear;
```

```

procedure Insert (List : in out List_Type;
 Item : in Element_Type) is
```

```

 Have_Duplicate : Boolean;
 Location : Positive;
begin
 if Full (List) then
 raise OVERFLOW;
 else
 -- Search for the correct location in the array
 Search_Array (The_Array => List.Items(1..List.Length),
 Key => Key_of (Item),
 Found => Have_Duplicate,
 Location => Location);
 if Have_Duplicate then
 raise DUPLICATE_KEY;
 else -- Insert the new item into the array
 -- Create space for the new element
 List.Items(Location+1..List.Length) := List.Items(Location..List.Length);
 List.Items(Location) := Item; -- put new element into the array
 List.Length := List.Length + 1; -- Update the length of the list
 end if;
 end if;
end Insert;
```

```

procedure Delete (List : in out List_Type;
 Key : in Key_Type) is
```

```

 Found : Boolean;
 Location : Positive;
begin
 -- Find the location of the element with the given key
 Search_Array (The_Array => List.Items(1..List.Length),
 Key => Key,
 Found => Found,
 Location => Location);
 if not Found then
 raise KEY_ERROR;
 else
 -- Remove the element by moving all following elements up in the array
 List.Items(Location..List.Length-1) := List.Items(Location+1..List.Length);
 List.Length := List.Length - 1; -- Update the length of the list
 end if;
end Delete;
```

```

procedure Modify (List : in out List_Type;
 Element : in Element_Type) is

 Found : Boolean;
 Location : Positive;
begin
 -- Find the location of the element
 Search_Array (The_Array => List.Items(1..List.Length),
 Key => Key_Of (Element),
 Found => Found,
 Location => Location);
 if not Found then
 raise KEY_ERROR;
 else
 -- Assertion: key = Key_Of (Element) (So key is not changed)
 List.Items (Location) := Element;
 end if;
end Modify;

```

```

procedure Retrieve(List : in List_Type;
 Key : in Key_Type;
 Element : Out Element_Type) is

 Found : Boolean;
 Location : Positive;
begin
 -- Find the location of the element with the given key
 Search_Array (The_Array => List.Items(1..List.Length),
 Key => Key,
 Found => Found,
 Location => Location);
 if not Found then
 raise KEY_ERROR;
 else
 Element := List.Items(Location);
 end if;
end Retrieve;

```

```

function Full (List : in List_Type) return Boolean is
begin
 return List.Length = List.Max_Size;
end Full;

```

```

function Empty (List : in List_Type) return Boolean is
begin
 return List.Length = 0;
end Empty;

```

```

function length (List : in List_Type) return Natural is
begin
 return List.Length;
end length;

procedure Traverse (List : in out List_Type) is
 Element : Element_Type; -- Use local variable to ensure keys are not changed
begin
 -- Each iteration: one element of the list is processed
 for Index in 1..List.Length loop
 Element := List.Items(Index); -- Make a local copy of the element
 Process (Element); -- Call user's procedure to process it
 -- Make sure the Key wasn't changed
 if Key-Of (Element) /= Key-Of (List.Items(Index)) then
 raise KEY_ERROR;
 else -- Put possibly modified element back into list
 List.Items(Index) := Element;
 end if;
 end loop;
end Traverse;

end Sorted_List;

```

### Linked List Implementations

In a linked list representation, the data elements are not constrained to be stored in sequential order; rather, the individual elements are stored somewhere in memory. The order of the element is maintained by *explicit* links between them. Figure 8-10 shows pictorially how a sorted list would be represented as a linked list. An empty list contains no nodes; the external pointer (`List` in Figure 8-10 is `null`).

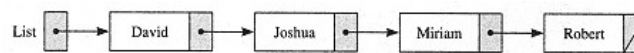


Figure 8-10  
Linked List

We construct the linked list structure much like the linked versions of the Stack and FIFO Queue ADTs. Each node contains both data and an explicit link to the next node in the list. Because the list is linear, the link field of the last node contains `null`, the special access value that says, "This doesn't designate anything." An external access variable (for example, `List` tells us where the list starts. You cannot access nodes in a linked list directly or randomly; that is, you cannot get directly to the twenty-fifth element in the list, as you can with a sequential array-based representation. Rather, you access the first node through the external pointer, the second node through the first node's "next" pointer, the third node through the second node's "next" pointer, and so on, to the end of the list.

In discussing the advantages of the linked stack and queue representations, we focused on the issue of dynamic storage allocation. Because the nodes were created dynamically, we did not have to declare an array variable large enough to store the maxi-



much number of elements, thus saving space when the list was smaller. There are, however, other advantages to a linked implementation unrelated to the issue of storage allocation. In fact, as you will see in Chapter, 10, a linked list not have to be stored in dynamically allocated space; we can store the elements in an array and link them explicitly using array indexes instead of access variables. So what's so great about linking?

Using a linked structure streamlines the algorithms that manipulate the elements. For instance, we can avoid all of the data movement that was necessary to insert and delete into a sequential list. With a linked representation, the

existing elements stay where they are as others are added and deleted. What changes are the links that establish the order of the elements.

### Implementing a Linked List

As the nodes used in the linked stack and linked queue, each node in a linked list contains two fields. The first is the `Info` field, which contains the list user's element. The second field, the `Next` field, contains the access value, or link, to the next node in the list.

In this chapter we implement a linked list whose nodes are dynamically allocated using allocator `new`. Here are the necessary declarations for the private part of Specification 8-1 (page 309).

```
private

 type Node_Type; -- Incomplete type declaration

 type List_Type is access Node_Type; -- Access to a node
 subtype Node_Ptr is List_Type; -- A synonym for our access type

 type Node_Type is -- Complete type declaration
 record
 Info : Element_Type; -- One element
 Next : Node_Ptr; -- Link to next node in the list
 end record;

end Sorted_List;
```

Because `List_Type` is no longer a discriminated record, the `Max_Size` discriminant in the public portion of the package declaration given earlier in the chapter must be removed. This deletion is the only change we need to make to the public portion of our package declaration.

Because Ada initializes every access variable to `null`, every list object we declare is initialized to empty. We can use the knowledge that an empty list is one whose external access value is null to write a short `Empty` function. The `Full` operation for a linked list is identical to the one we used for a linked stack and a linked queue. We try to allocate new memory and see if `STORAGE_ERROR` is raised. If it is raised, the list is full. Otherwise the list is not full and we can deallocate the memory just obtained.

The `Clear` operation for a linked list is more complicated than its sequential list counterpart, since the dynamically allocated space used by the elements must be freed. The easiest approach is to unlink each successive node in the list and free it. Because this algorithm is exactly like the clear operation for stacks developed in Chapter 6, we'll reuse the code, changing "Stack" references to "List."

### The Traverse Operation

Early in this chapter, the algorithm for the `Traverse` operation that we developed for the sequential list used an array index to access each element in the array used to store the list's elements. A for loop allowed us to easily move from one element to the next. In the linked implementation, we must use the `Next` field of a node to locate the next node in the list. How do we know when we have processed all of the elements in the list? As the end of the list is marked by a null value, we stop when we reach `null`. Here is the algorithm:

#### Traverse

```

Location := List; -- Location designates 1st node of list
loop
 exit when Location is null;
 Process (Location.all.Info); -- Process info in node designated by Location
 Location := Location.all.Next; -- Advance Location to the next node
end loop;
```

The implementation has changed, but the basic algorithm to traverse a list stayed the same. Trace this algorithm using the list shown in Figure 8-10. Does this algorithm work correctly when the list is empty? Of course we still need to add the logic to detect key changes that we have developed for our sequential list traverse operation.

### The Length Operation

Our linked list `List_Type` does not contain a `Length` field as our sequential one did. To determine the number of elements in the list we must traverse the list counting each node as we go. The algorithm is nearly identical to the one we developed for the traverse operation.

#### Length

```

Count := 0
Location := List
loop
 exit when Location is null;
 Count := Count + 1;
 Location := Location.all.Next;
end loop;
return Count
```

### Finding a List Element

The rest of the List operations (retrieve, modify, insert, and delete) require searching the list to find the element to process. As we did for the sequential list implementation, we write a procedure to simplify these other operations. This time we call it `Search_Linked_List`. This procedure takes an access value that designates the list and a key value as inputs and searches the list for the element with the specified key. If the element is found, the output parameter `Location` (an access variable) designates the node with the desired key; otherwise, `Location` designates the node that would come immediately after the node containing the desired key, if that key were in the list. We are going to add another `out` parameter to this procedure. In the linked list implementation, an access variable that designates the node that precedes the one designated by `Location` is useful both in inserting and in deleting elements. We will call this parameter `Pred_Loc`. Specification 8-4 defines this search procedure.

#### Specification 8-4— Linked List Search

```

procedure Search_Linked_List (List : in List_Type;
 Key : in Key_Type;
 Found : out Boolean;
 Pred_Loc : out Node_Ptr;
 Location : out Node_Ptr);
-- Searches for the location of Key in the List

-- Preconditions : The nodes in List are in ascending order

-- Postconditions : If Key is in List
-- Found is True
-- Location designates the element with Key
-- Pred_Loc designates the predecessor of the node designated by
-- Location. If Location designates the first node in the
-- list, Pred_Loc is null.
-- else
-- Found is False
-- Location designates the node that would follow a node containing
-- Key. If Key's node would follow the last node in the
-- list, Location is null.
-- Pred_Loc designates the predecessor of the node designated by
-- Location. If Location designates the first node in the
-- list, Pred_Loc is null.

```

The algorithm is similar to the sequential list version of this procedure: We start at the beginning of the list and loop until (1) we have found an element with the desired key, (2) we have passed all the elements in the list whose keys are smaller than *Key*, or (3) we have reached the end of the list. Here is the basic loop structure using a local variable, *Current*, that designates the node we currently are looking at. Notice the order of terms in the exit condition and the use of the short-circuit Boolean operators to prevent *CONSTRAINT\_ERROR* from being raised when we reach the end of the list.

```

Current := List; -- Start at the beginning of the linked list
loop -- Each iteration, one node is checked
 exit when Current = null or else -- Reached the end
 Key < Key_Of (Current.all.Info) or else -- Passed by
 Key = Key_Of (Current.all.Info); -- Found it
 Current := Current.all.Next; -- Go on to next element
end loop;
Location := Current;
Found := Current /= null and then Key = Key_Of (Current.all.Info);

```

Figure 8-11 shows the different cases indicated in the postconditions for procedure *Search\_Linked\_List*. Setting *Location* is easy. It is given the value of *Current* determined by our loop. But setting *Pred\_Loc* is a problem. In the sequential list implementation, this operation would be a simple task. Given the location (index) of an array slot, it is easy to figure out its predecessor—we merely set *Pred\_Loc* to *Index - 1*.

Given the designator to a node in a linked list, however, we cannot figure out the location of its predecessor in the list. The links go in only one direction.

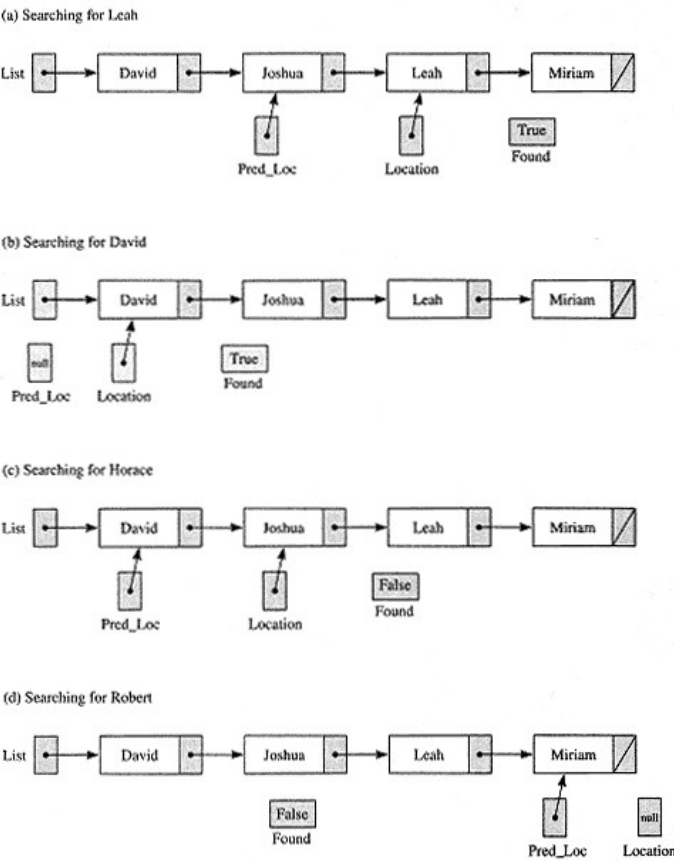
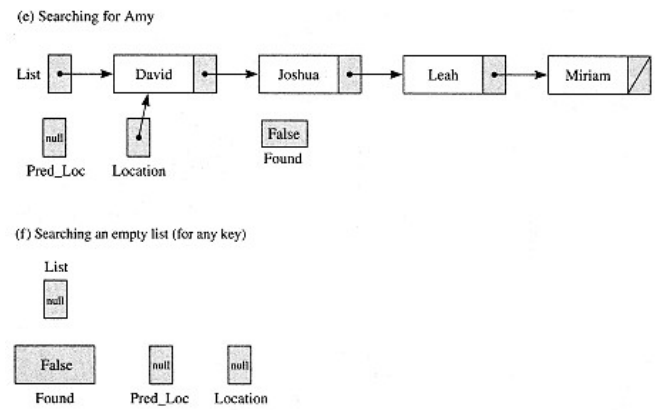


Figure 8-11  
Searching a Linked List



A solution to this problem is to use a *pair* of pointers to traverse the list. As *Current* moves along the list, searching for a key, another access variable, *Previous*, trails behind by one node. Using a second access variable complicates our search loop slightly. We initialize *Current* to *List* and *Previous* to *null*. Then, inside the loop, we advance both pointers. As Figure 8-12 shows, the process resembles the movement of an inchworm. *Previous* (the tail of the inchworm) catches up with *Current* (the head), and then *Current* advances.

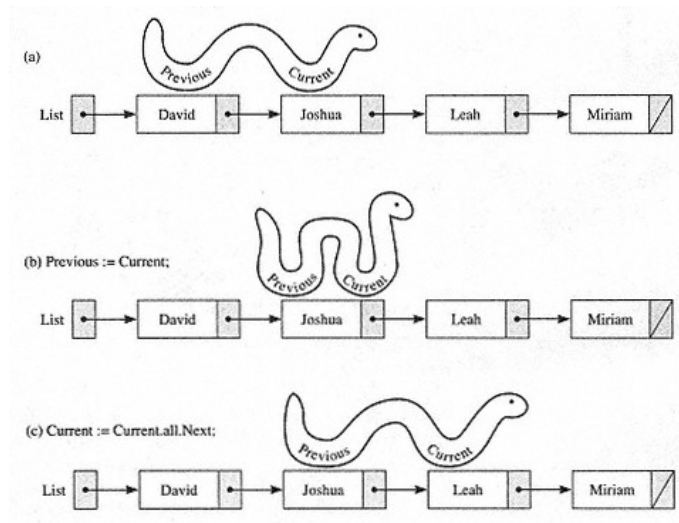


Figure 8-12  
The Inchworm Effect

Letting Previous trail Current in searching the list makes setting the out parameters easy. Location is set to Current and Pred\_Loc is set to Previous as shown in the following revised loop structure. The code for the complete procedure is shown in Body 8-3 (page 354).

```
Current := List; -- Start at the beginning of the linked list
Previous := null; -- No predecessor for 1st element in the list
loop -- Each iteration, one node is checked
 exit when Current = null or else -- Reached the end
 Key < Key_Of (Current.all.Info) or else -- Passed by
 Key = Key_Of (Current.all.Info); -- Found it
 Previous := Current; -- Move inchworm's tail
 Current := Current.all.Next; -- Move inchworm's head
end loop;
```

```
Location := Current;
Pred_Loc := Previous;
Found := Current /= null and then Key = Key_Of (Current.all.Info);
```

Procedure Search\_Linked\_List is internal to the list ADT; it is to be used only by the list operations. The application programmer should never be able to access list data through the pointers; rather, the access mechanism is encapsulated by the list operations. With the help of procedure Search\_Linked\_List, the rest of the list operations are very simple to write.

### The Retrieve and Modify Operations

The basic algorithms for Retrieve and Modify are identical to the sequential list versions given earlier in the chapter. This time our search returns a designator to the node containing the desired element rather than an array index.

#### Retrieve

```
Search Linked List
if Found then
 Element := Location.all.Info
else
 raise KEY_ERROR
end if
```

#### Modify

```
Search Linked List
if Found then
 Location.all.Info := Element
else
 raise KEY_ERROR
end if
```

### The Insert Operation

To insert a new element into the linked list, we use the general algorithm developed for the sequential list implementation:

## Insert

```

Find the place where the new element belongs
Create space for the new element
Put the new element into the list

```

This algorithm is illustrated in Figure 8-13. The first task is simple: we call `Search_Linked_List`. Of course, we do not expect to find the new element's key in the list. If we do, we need to raise `DUPLICATE_KEY`.

The second task is to create space for the new element. In the sequential version of `Insert`, we had to move all the data elements down one slot, from the insertion location to the end of the list, to make room for the new element. One of the advantages to a linked list implementation is that all this data movement can be avoided, because the space for the new element does not come out of the existing structure. Instead we create space dynamically at run time by asking the system to allocate some free memory space, using the allocator `new`. We then can put the value of `Element` into the Info part of the new node.

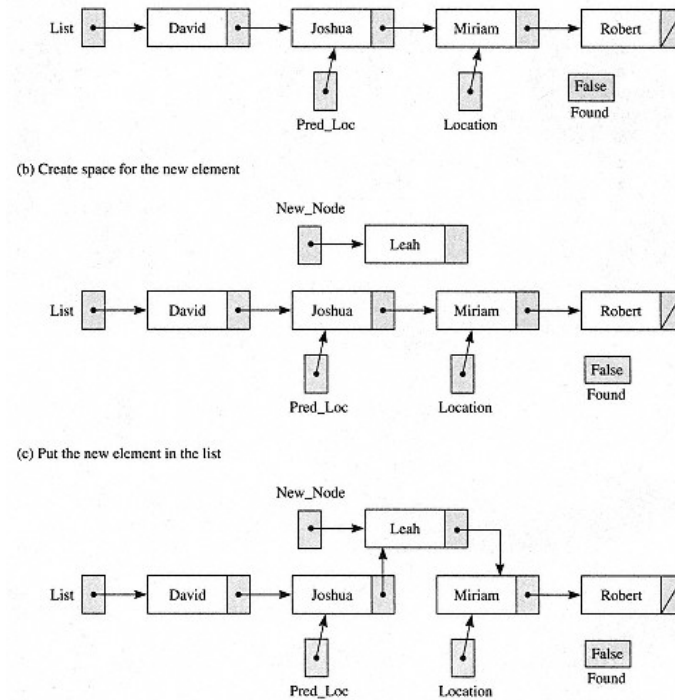


Figure 8-13  
Inserting into a Linked List

### Create space for the new element

```
New_Node := new Node_Type; -- get the space
New_Node.all.Info := Element; -- assign the new element
```

The last task of the insert algorithm is to put the new element in the list. To complete the insertion, we must manipulate the access variables to link the new node into its appropriate position in the list. There are four cases to consider: (1) inserting into an empty list; (2) inserting before the first node in the list; (3) inserting into the "middle" of the list; and (4) inserting after the last node in the list.

In the first case, we merely need to set `List` (the external pointer to the list) to designate the new node [Figure 8-14(a)]. As with all access variables, the `Next` field in the new node is initialized to `null` when the node comes into existence.

### Inserting into an Empty List

```
List := New_Node
```

The second case is similar: We set `New_Node`'s `Next` field to designate to the first list node and then set `List` to point to the new node [Figure 8-14(b)].

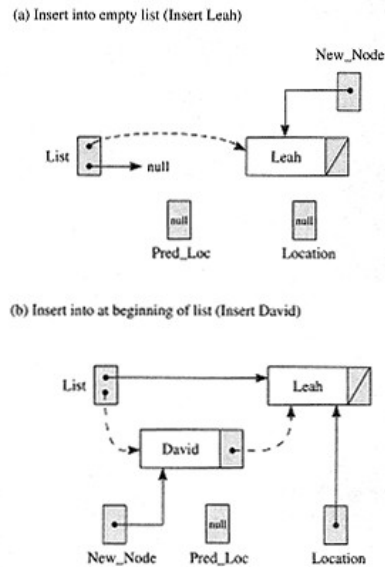
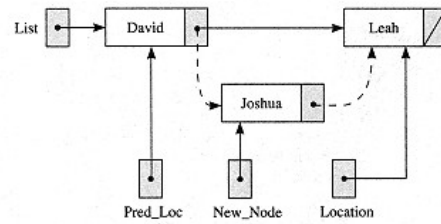


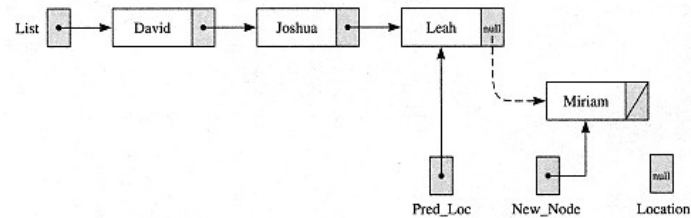
Figure 8-14  
Four Insert Cases. Solid arrows show the values of pointers just after obtaining memory for the new node. Dashed arrows show the values of pointers after the insert operation is complete.



(c) Insert in middle of list (Insert Joshua)



(d) Insert at end of list (Insert Miriam)



### Inserting at the Front of the List

```
New_Node.all.Next := List
List := New_Node
```

Actually, because `List = null` when the list is empty, the first case is the same as the second, so we combine them.

In the third case (inserting into the "middle" of the list), we link the new node into place after its predecessor), as shown in Figure 8-14(c).

### Inserting into the Middle of the List

```
New_Node.all.Next := Location
Pred_Location.all.Next := New_Node
```

In the fourth case (inserting at the end of the list), we link the new node after the last node in the list as illustrated in Figure 8-14(d). Again, the `Next` field in the new node is initialized to `null` when that node comes into existence.

### Inserting at the End of the List

```
Pred_Location.all.Next := New_Node
```

Because `Location` is equal to `null` when we are inserting at the end of the list, the third and fourth cases can be combined. The whole algorithm for the Insert operation is shown below.

### Insert

```

Search_Linked_List -- Find the insert location
New_Node := new Node_Type -- Create Space for the new element
New_Node.all.Info := Element
if the list is empty or inserting at the beginning of the list
 New_Node.all.Next := List
 List := New_Node
else
 New_Node.all.Next := Location
 Pred_Location.all.Next := New_Node
end if

```

How do we know if the list is empty or if we are inserting at the beginning of the list? In either of these cases, the value of `Pred_Loc` returned from `Search_Linked_List` is `null`.

There is one further refinement we can make to this algorithm. We can supply an initial value for a node when we allocate memory for it. We can use this feature to eliminate the need for the local variable `New_Node` and to reduce the size of our procedure as shown in the following version of our insert algorithm:

### Insert

```

Search_Linked_List
if Pred_Loc = null -- Inserting at the beginning
 List := new Node_Type'(Info => Element, Next => List)
else
 -- Inserting at middle or at the end
 Pred_Location.all.Next := new Node_Type'(Info => Element, Next => Location)
end if

```

### The Delete Operation

The last operation specified for our list class is the deletion of an element with a given key. The basic tasks for the delete algorithm are

#### Delete

```

Find the element in the list
Remove the element from the list

```

Procedure `Search_Linked_List` gives us a designator to the node we wish to delete, as well as one that designates its predecessor, if there is one.

The second task, to remove the element from the list, has two parts. First we must "unlink" the deleted element from the list; then we dispose of its node space. When unlinking the deleted element, we must pay attention to the node's position in the list. We consider four cases: (1) deleting the only list element, leaving the list empty; (2) deleting the first node; (3) deleting a "middle" node; and (4) deleting the last node.

In the first case, we merely set `List` to `null`, as pictured in Figure 8-15(a). In the second case (deleting the first node in the list), we set `List` to `Location.all.Next`, as shown in Figure 8-15(b). Because `Location.all.Next = null` when the delete node

is the only node in the list, we can combine these two cases in the code. We can detect both cases by checking `Pred_Loc` for a null value.

In the third case (deleting a middle node), we must make the link from the preceding node "jump over" the deleted node [Figure 8-15(c)] by assigning the value of `Location.all.Next` to `Pred_Loc.all.Next`.

In the fourth case (deleting the last node), we merely set `Pred_Loc.all.Next` to null, as pictured in Figure 8-15(d). Because `Location.all.Next` is null when the delete node is at the end of the list, we can combine the third and fourth cases.

After unlinking the node designated by `Location`, we recycle its space with a call to a procedure instantiated from `Unchecked_Deallocation`.

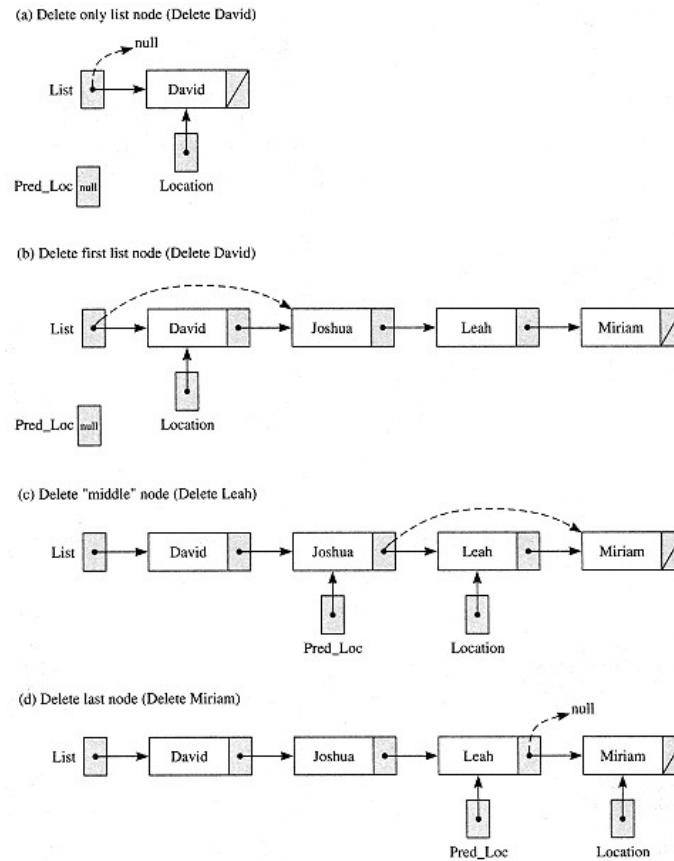


Figure 8-15

Deleting from a Linked List. Solid arrows show the values of pointers just after the search. Dashed arrows show the values of pointers after the delete operation is complete.

## Linked List Package Body

Body 8-3 contains the complete body for our linked implementation of a sorted list.

### Body 8-3—

#### Linked List Implements Specification 8-1

```
with Unchecked_Deallocation;
package body Sorted_List is
 -- Instantiate procedure for recycling node memory
 procedure Free is new Unchecked_Deallocation (Object => Node_Type,
 Name => Node_Ptr);

 -- Local procedures

 procedure Search_Linked_List (List : in List_Type;
 Key : in Key_Type;
 Found : out Boolean;
 Pred_Loc : out Node_Ptr;
 Location : out Node_Ptr) is
 -- Searches for the location of Key in the List

 -- Preconditions : The nodes in List are in ascending order
 --
 -- Postconditions : If Key is in List
 -- Found is True
 -- Location designates the element with Key
 -- Pred_Loc designates the predecessor of the node designated
 -- by Location. If Location designates the first
 -- node in the list, Pred_Loc is null.
 --
 -- else
 -- Found is False
 -- Location designates the node that would follow a node
 -- containing Key. If Key's node would follow
 -- the last node in the list, Location is null.
 -- Pred_Loc designates the predecessor of the node designated
 -- by Location. If Location designates the first
 -- first node in the list, Pred_Loc is null.

 Current : Node_Ptr; -- Local copy of Location
 Previous : Node_Ptr; -- Local copy of Pred_Loc
 begin
 Current := List; -- Start at the beginning of the linked list
 Previous := null; -- No predecessor for 1st element in the list
 loop -- Each iteration, one node is checked
 exit when Current = null or else -- Reached the end
 Key < Key_Of (Current.all.Info) or else -- Passed by
 Key = Key_Of (Current.all.Info); -- Found it
 Previous := Current; -- Move inchworm's tail
 Current := Current.all.Next; -- Move inchworm's head
 end loop;
 end Search_Linked_List;
end Sorted_List;
```

```

end loop;
-- Copy results to out parameters
Location := Current;
Pred_Loc := Previous;
Found := Current /= null and then Key = Key_Of (Current.all.Info);
end Search_Linked_List;

-- Operations

procedure Clear (List : in out List_Type) is
 To_Recycle : Node_Ptr; -- Designates a node for recycling
begin
 loop
 exit when List = null; -- Exit when list is empty
 To_Recycle := List; -- Unlink the
 List := List.all.Next; -- the first node
 Free (To_Recycle); -- Recycle the node
 end loop;
end Clear;

procedure Insert (List : in out List_Type;
 Item : in Element_Type) is

 Have_Duplicate : Boolean;
 Pred_Loc : Node_Ptr;
 Location : Node_Ptr;
begin
 -- Search for the correct location in the linked list
 Search_Linked_List (List => List,
 Key => Key_Of (Item),
 Found => Have_Duplicate,
 Pred_Loc => Pred_Loc,
 Location => Location);
 if Have_Duplicate then
 raise DUPLICATE_KEY;
 elsif Pred_Loc = null then
 List := new Node_Type'(Info => Item, Next => List); -- Add at beginning
 else -- Add at the middle or end
 Pred_Loc.all.Next := new Node_Type'(Info => Item, Next => Location);
 end if;
exception
 when STORAGE_ERROR => -- Overflow when no storage available
 raise OVERFLOW;
end Insert;

procedure Delete (List : in out List_Type;
 Key : in Key_Type) is

 Found : Boolean;
 Pred_Loc : Node_Ptr;
 Location : Node_Ptr;

```

```

begin
 -- Find the location of the element with the given key
 Search_Linked_List (List => List,
 Key => Key,
 Found => Found,
 Pred_Loc => Pred_Loc,
 Location => Location);

 if not Found then
 raise KEY_ERROR;
 elsif Pred_Loc = null then
 List := Location.all.Next; -- Delete first element
 else
 Pred_Loc.all.Next := Location.all.Next; -- Delete middle or last element
 end if;
 Free (Location); -- Recycle node memory
end Delete;

```

```

procedure Modify (List : in out List_Type;
 Element : in Element_Type) is

```

```

 Found : Boolean;
 Pred_Loc : Node_Ptr;
 Location : Node_Ptr;
begin
 -- Find the location of the element
 Search_Linked_List (List => List,
 Key => Key_Of (Element),
 Found => Found,
 Pred_Loc => Pred_Loc,
 Location => Location);

 if not Found then
 raise KEY_ERROR;
 else
 Location.all.Info := Element;
 end if;
end Modify;

```

```

procedure Retrieve (List : in List_Type;
 Key : in Key_Type;
 Element : out Element_Type) is

```

```

 Found : Boolean;
 Pred_Loc : Node_Ptr;
 Location : Node_Ptr;
begin
 -- Find the location of the element with the given key
 Search_Linked_List (List => List,
 Key => Key,
 Found => Found,
 Pred_Loc => Pred_Loc,
 Location => Location);

```

```

 if not Found then
 raise KEY_ERROR;
 else
 Element := Location.all.Info;
 end if;
end Retrieve;

```

```

function Full (List : in List_Type) return Boolean is
 New_Node : Node_Ptr; -- Designates a temporary node
begin
 New_Node := new Node_Type; -- Get memory for a new node
 Free (New_Node); -- Now recycle the memory
 return False; -- There is still memory available
exception
 when STORAGE_ERROR => -- When there is no memory available
 return True; -- the list is full
end Full;

```

```

function Empty (List : in List_Type) return Boolean is
begin
 return List = null;
end Empty;

```

```

function Length (List : in List_Type) return Natural is
 Location : Node_Ptr; -- Designates current node
 Count : Natural;
begin
 Count := 0;
 Location := List;
 loop
 exit when Location = null;
 Count := Count + 1;
 Location := Location.all.Next;
 end loop;
 return Count;
end Length;

```

```

procedure Traverse (List : in out List_Type) is
 Element : Element_Type; -- Use local variable to ensure keys are not changed
 Location : Node_Ptr; -- Designates current node
begin
 Location := List;
 loop
 exit when Location = null;
 Element := Location.all.Info; -- Make a local copy of the element
 Process (Element); -- Call user's procedure to process it
 -- Make sure the key wasn't changed
 if Key_Of (Element) /= Key_Of (Location.all.Info) then
 raise KEY_ERROR;
 end if;
 Location := Location.all.Next;
 end loop;
end Traverse;

```

```

 else -- Put possibly modified element back into list
 Location.all.Info := Element;
 end if;
 Location := Location.all.Next; -- Move to next node
end loop;
end Traverse;

end Sorted_List;

```

### Analyzing the List Implementations

Now let's compare the sequential and linked implementations of the list ADT. Just as when we compared stack and queue implementations, we look at several different factors: the length of the source code required to implement the solutions, the amount of memory required to store the structure, and the amount of "work" the solution does.

The table that follows summarizes the size of the source code of the declarations, procedures, and functions for each implementation. As you can see from the chart, the linked implementation takes almost 35% more lines of code than the sequential implementation. Deciding which design is easier to understand is very subjective. Which implementation seems simpler to you?

**Comparison of Executable Source Lines of Code**

|                               | <i>Sequential<br/>Implementation</i> | <i>Linked<br/>Implementation</i> |
|-------------------------------|--------------------------------------|----------------------------------|
| Declarations (Private Part)   | 6                                    | 8                                |
| Clear                         | 4                                    | 10                               |
| Empty                         | 4                                    | 4                                |
| Full                          | 4                                    | 10                               |
| Length                        | 4                                    | 13                               |
| Search (Array or Linked List) | 16                                   | 21                               |
| Insert                        | 21                                   | 22                               |
| Delete                        | 16                                   | 20                               |
| Modify                        | 15                                   | 17                               |
| Retrieve                      | 16                                   | 18                               |
| Traverse                      | 13                                   | 17                               |
| Total                         | 119                                  | 160                              |

Now let's compare the storage requirements. The array field of our list record takes the same amount of memory, no matter how many array slots actually are used, because we need to reserve space for the maximum possible. The linked implementation using dynamically allocated storage space requires only enough space for the number of elements actually in the list at run time. However, as we discussed in detail in Chapter 7, each node element is larger, because we must store the link (the `Next` field) as well as the user's data.

We also can compare the relative "efficiency" of the two implementations. Most of the operations are nearly identical in the two implementations. The `Full` and `Empty` func-



tions in both implementations are clearly  $O(1)$ . As in the stack and queue operations, `Clear` is a  $O(1)$  operation for a sequential list but becomes a  $O(N)$  operation for a linked list. The sequential implementation merely marks the list as empty, whereas the linked implementation must access each list element to free its dynamically allocated space. The traversal operation processes each element in the list once. `Traverse` is  $O(N)$  for both list representations.

Both implementations of the local search procedure (`Search_Array` and `Search_Linked_List`) use sequential searches. Beginning at the first element, they examine one element after another until the correct element (or place) is found. Because they potentially must search through all the elements in a list, the loops in both procedures are  $O(N)$ . The code to set the output parameters is  $O(1)$ . Thus both searches are  $O(N)$ . This Big-O analysis does not mean that they take the same amount of time to execute. The linked list version reassigns two pointers in each iteration of the loop body (the search pointer `Current` and the trailing `Previous`), whereas the sequential list version only increments an index. Thus we might expect the sequential version to be a little quicker but by less than one order of magnitude. (The linked version might take 50% longer, but not 10 times longer to execute.)

The `Retrieve` and `Modify` procedures are virtually identical for the two implementations. They first call the search [ $O(N)$ ] and then assign the output parameters [ $O(1)$ ]. Since  $O(N) + O(1) = O(N)$ , these operations are  $O(N)$  in both implementations.

In both list implementations, the insert operations call the local search procedure to find the insertion position; therefore, the search parts of these algorithms are  $O(N)$ . The sequential list also must move down all the elements that follow the insertion position to make room for the new element. The number of elements to be moved by the array slice assignment statement ranges from zero, when we insert to the end of the list, to `List.Length`, when we insert to the beginning of the list. Here is a case where an assignment statement is *not*  $O(1)$ , but  $O(N)$ . So the insertion part of the algorithm is also  $O(N)$  for the sequential list. Because  $O(N) + O(N) = O(N)$ , the sequential list's `Insert` procedure is  $O(N)$ .

The insertion part of the algorithm for the linked list representation simply requires the reassignment of a couple of pointers—an  $O(1)$  operation. This characteristic is one of the main advantages of linking. However, adding the insertion task to the search task gives us  $O(N) + O(1) = O(N)$ —the same Big-O approximation as for the sequential list! Doesn't the linking offer any advantage in efficiency? Remember that the Big-O evaluations are only rough approximations of the amount of work that an operation does. The next section presents a timed test to see whether the linked list's insertion operation really is faster.

The `Delete` procedure is similar to `Insert`. In both implementations, the search task is an  $O(N)$  operation. Then the sequential list's insert operation "deletes" the element by moving up all the subsequent elements in the list, which adds  $O(N)$ . The whole procedure is  $O(N) + O(N)$ , or  $O(N)$ . The linked list deletes the element by unlinking it from the list and calling `Free`, which adds  $O(1)$  to the search task. The whole procedure is  $O(N) + O(1)$ , or  $O(N)$ . Thus both `Delete` procedures are  $O(N)$ ; for large values of  $N$ , they are roughly equivalent.

This Big-O analysis does not mean that they will take the same amount of time to execute, however. The sequential implementation requires a great deal of data movement for both `Insert` and `Delete`. Does all this data movement really make any difference? It doesn't matter too much when the list is very small. If there are thousands of elements, however, the data movement starts to add up.

The table that follows summarizes the Big-O comparison of the list operations for sequential and linked implementations.

| <b>Big-O Comparisons of List Operations</b> |                                 |                                  |
|---------------------------------------------|---------------------------------|----------------------------------|
|                                             | <i>Array<br/>Implementation</i> | <i>Linked<br/>Implementation</i> |
| Clear                                       | <b>O(1)</b>                     | <b>O(N)</b>                      |
| Empty                                       | O(1)                            | O(1)                             |
| Full                                        | O(1)                            | O(1)                             |
| Length                                      | <b>O(1)</b>                     | <b>O(N)</b>                      |
| Search (Array or Linked List)               | O(N)                            | O(N)                             |
| Insert                                      |                                 |                                  |
| Search O(N) +                               | <b>O(N)</b>                     | <b>O(1)</b>                      |
| Total                                       | O(N)                            | O(N)                             |
| Delete                                      |                                 |                                  |
| Search O(N) +                               | <b>O(N)</b>                     | <b>O(1)</b>                      |
| Total                                       | O(N)                            | O(N)                             |
| Modify                                      |                                 |                                  |
| Search O(N) +                               | O(1)                            | O(1)                             |
| Total                                       | O(N)                            | O(N)                             |
| Retrieve                                    |                                 |                                  |
| Search O(N) +                               | O(1)                            | O(1)                             |
| Total                                       | O(N)                            | O(N)                             |
| Traverse                                    | O(N)                            | O(N)                             |

Let's look at an actual timed test case, with real numbers. The following table shows how long it takes one particular computer to insert and then delete 1000 elements (a record containing an integer key and a 200 character string) in various orders. The test program ran three cases:

1. *Random inserts/deletes.* The test program inserted the elements in "random" order, then deleted them in the same order.
2. *End of list inserts/deletes.* The test program inserted elements with keys from 1 through 1000, in order; then it deleted the elements in reverse order (1000 through 1) so that insertions and deletions always took place at the end of the list.
3. *Front of list inserts/deletes.* The test program inserted elements in reverse order, from key 1000 through 1; then it deleted the elements in order (1 through 1000) so that insertions and deletions always took place at the beginning of the list.

| <b>Results of the Timed Implementation Comparisons (in seconds)</b> |                                 |                                  |
|---------------------------------------------------------------------|---------------------------------|----------------------------------|
|                                                                     | <i>Array<br/>Implementation</i> | <i>Linked<br/>Implementation</i> |
| <b>Case 1: Random</b>                                               |                                 |                                  |
| Insert                                                              | 1.98                            | 0.71                             |
| Delete                                                              | 1.95                            | 0.67                             |

(table continues on next page)

**Results of the Timed Implementation Comparisons (in seconds) (cont.)**

|                           | <i>Array<br/>Implementation</i> | <i>Linked<br/>Implementation</i> |
|---------------------------|---------------------------------|----------------------------------|
| <b>Case 2: List End</b>   |                                 |                                  |
| Insert                    | 1.64                            | 1.27                             |
| Delete                    | 1.57                            | 1.28                             |
| <b>Case 3: List Start</b> |                                 |                                  |
| Insert                    | 2.46                            | 0.03                             |
| Delete                    | 2.46                            | 0.02                             |

As you can see from the information in the table, the linked implementation is faster for random additions and deletions to the list (case 1). This result is as we expected, due to the data movement required in the sequential list implementation.

In case 2, we minimize the data movement by inserting and deleting only at the end of the list. The search task requires  $N$  comparisons, which means we have to search the whole list. However, the create/delete space task is  $O(1)$  for both implementations, because no elements must be moved. Although case 2 is the *best* case for data movement in the sequential list, on this particular computer it still is not as fast as the linked version. With an array, the address of an element must be calculated from an index (remember the formulas we developed in Chapter 3?). The addresses are directly available in an access value; no calculations are necessary. The larger times for the linked implementation compared to case 1 reflect that in this case our operations must search the *entire* list.

If we maximize the data movement by inserting and deleting only at the front of the list (case 3), the search task becomes  $O(1)$ ; we always find the place with one comparison. The linked version's create/delete space task is  $O(1)$  as always, resulting in  $O(1)$  Insert and Delete operations. However, case 3 is the *worst* case for data movement in the sequential list; all the elements in the list must be moved down for each insertion, and they all must be moved up for each deletion. The results of the timed test reflect this disparity: The sequential list took about *100 times as long* as the linked list to perform the same operations.

Though we cannot avoid the data movement in the sequential implementation, we can improve the searching task by replacing the sequential search with the binary search we developed in Chapter 2. This modification improves the "search" part of the algorithm from  $O(N)$  to  $O(\log_2 N)$ . We cannot make a similar improvement, however, for searching the linked list, because we cannot access the elements randomly; we only can access an element through its predecessor in the list. Chapter 12 discusses the binary search tree, a linked structure that supports  $O(\log_2 N)$  searches.

**Other Factors to Consider**

Choosing to use a sequential or linked implementation for a particular application depends greatly on what you want to do with the data stored in the list. We already have compared the operations that our sorted list package specifies. There are other operations that might be specified for an ordered list, some of which are better for one implementation than another. For instance, if the dominant activity in your application is to access the  $N$ th element in the list, or if you need to print out the elements in reverse order, a sequential list is preferable to a linked list. On the other hand, if the maximum number of elements in the list cannot be predicted, a linked list is preferable, especially if the list elements for the application are large.

## Testing the List Operations

By means of a test driver, we can test the list package independently of the program that will use it. The simplest kind of test driver sets up a scenario that executes the various test cases. To make sure that you test all the necessary cases, you should make a test plan, listing all the list operations and the tests each needs.

You can use the same test driver to test both the sequential and linked list implementations; we merely replace the list object declaration. Except for the `Full` function, you should get the same test results for both implementations.

You should test the generic package with two different element types; a record type like that used in the address book application and a simple type-like integer. For the sequential implementation, you also can set `Max_Size` to a relatively small number for your tests. Knowing that the code works the same whether `Max_Size` is 5 or 1000, you can set `Max_Size` to 5 for the test.

## Summary

We have seen how lists may be represented in a sequential or linked representation and how they may be implemented with static (an array) or dynamic variables. The list class specification at the beginning of this chapter didn't mention any of these design issues, so we were free to implement the abstract data structure list in many ways. Nothing in the specification of this class said that the list was sequential or linked, or that its elements were stored in statically or dynamically allocated storage.

We could specify a number of other operations for the list package. Some operations, such as one to find the preceding node in a list, are easy to implement for a sequential list but would be difficult to implement using a list that is linked in one direction (like the lists in this chapter). This operation would be simpler if the list had links going both forward and backward. We can think of many variations for representing a linked list in order to simplify the kinds of operations that are specified for the list: doubly linked lists, circular lists, lists that are accessed from both the beginning and the end. We continue this discussion in Chapter 9.

The idea of linking the elements in a data structure is not specific to the type of list we have discussed in this chapter. In Chapters 6 and 7 we implemented stacks and queues as linked structures; we use this powerful tool to implement other data structures in this book.

In Chapters 9 and 10 we look at several other alternative implementations of a sorted list. The change of implementation, however, does not change the specification of our list operations, and for the most part, does not even change their general algorithms.

## Exercises

Use the following declarations in Exercises 1 and 2:

```
type Credit_Rating_Type is (Poor, Fair, Good, Excellent);
type String20 is String (1..20);
```

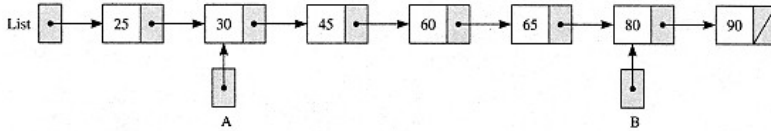
```
type Account_Rec is
 record
 Account_Number : Positive;
 Company_Name : String20;
 Balance_Due : Float;
 Credit_Rating : Credit_Rating_Type;
 end record;
```

1. (a) Write a function called `Number_Of` that returns the `Account_Number` field of a parameter of type `Account_Rec`.  
 (b) Instantiate package called `Account_List` for a sorted list class whose elements are `Account_Recs` ordered by `Account_Number`.  
 (c) Instantiate an iterator that displays the names of all companies in the list that have an Excellent credit rating. You need to write the procedure that is used as the actual generic parameter in this instantiation.
2. (a) Write a function called `Name_Of` that returns the `Company_Name` field of a parameter of type `Account_Rec`.  
 (b) Instantiate package for a sorted list class whose elements are `Account_Recs` ordered by `Company_Name`.  
 (c) Instantiate an iterator that displays the account numbers and names of all companies in the list that have an account balance greater than \$1000. You need to write the procedure that is used as the actual generic parameter in this instantiation.  
 (d) Instantiate an iterator that subtracts \$100 from the `Balance_Due` field of every company in the list. You need to write the procedure that is used as the actual generic parameter in this instantiation.
3. Rewrite the `Label` procedure given in the address book application developed in this chapter so that it prints labels sorted by zip code instead of by name. Hint: instantiate a second sorted list package using the zip code as a key.
4. (a) Explain the difference between a sequential and a linked representation of a list.  
 (b) Give an example of a problem for which a sequential list would be the better solution.  
 (c) Give an example of a problem for which a linked list would be the better solution.
5. *True or False?* If you answer False, correct the statement.
  - (a) An array is a random-access structure.
  - (b) A sequential list is a random-access structure.
  - (c) A linked list is a random-access structure.
  - (d) A sequential list is always stored in a static variable.
  - (e) The elements in a linked list can be stored in an array.
6. In relation to elaboration/execution time, what is meant by
  - (a) a static object?
  - (b) a dynamic object?

Use the linked list pictured below in Exercises 7-10. The declaration for the nodes shown in this figure is

```
type Node_Type;
type Node_Ptr is access Node_Type;
type Node_Type is
 record
 Info : Natural;
 Next : Node_Ptr;
 end record;
```

```
List : Node_Ptr;
A : Node_Ptr;
B : Node_Ptr;
```



7. Give the values of the following expressions:

- (a) A.all.Info
- (b) B.all.next.all.Info
- (c) List.all.Next.all.Next.all.Info

8. Are the following statements True or False?

- (a) List.all.Next = A
- (b) A.all.Next.all.Info = 60
- (c) B.all.Next = null
- (d) List.all.Info = 25

9. Decide whether the *syntax* in the following statements is valid or invalid. If it is valid, mark it OK; if it is invalid, explain what is wrong.

- (a) List.all.Next := A.all.Next;
- (b) List.all.Next := B.all.Next.all;
- (c) List.all := B;
- (d) B := A.all.Next.all.Info;
- (e) A.all.Info := B.all.Info;
- (f) List := B.all.Next.all.Next;
- (g) B := B.all.Next.all.Next.all.Next;

10. Write a statement to do each of the following:

- (a) Make List designate the node containing 45.
- (b) Make B designate the last node in the list.
- (c) Make List designate an empty list.
- (d) Set the Info value of the node containing 45 to 46.

Use the following declarations in Exercises 11-16.

```

type Node_Type;
type Node_Ptr is access Node_Type;
type Node_Type is
 record
 Info : Natural;
 Next : Node_Ptr;
 end record;

```

```

List : Node_Ptr;
Ptr : Node_Ptr;

```

11. Show what is written by the following segment of code:

```

List := new Node_Type;
Ptr := new Node_Type;
List.all.Info := 2;
Ptr.all.Info := 5;
List := Ptr;
Ptr.all.Info := 7;
Integer_IO.Put (Ptr.all.Info);
Integer_IO.Put (List.all.Info);

```

12. Show what is written by the following segment of code:

```

List := new Node_Type'(Info => 10, Next => null);
Ptr := new Node_Type'(Info => 18, Next => null);
Ptr := new Node_Type;
Ptr.all.Info := 20;
Ptr.all.Next := List;
List := Ptr;
loop
 exit when Ptr = null;
 Integer_IO.Put (Ptr.all.Info);
 Text_IO.New_Line;
 Ptr := Ptr.all.Next;
end loop;

```

13. Show what is written by the following segment of code:

```

List := new Node_Type'(Info => 0, Next => null);
for Count in 1..4 loop
 List := new Node_Type'(Info := Count, Next := List);
end loop;
Ptr := List;
loop
 exit when Ptr = null;
 Integer_IO.Put (Ptr.all.Info);
 Text_IO.New_Line;
 Ptr := Ptr.all.Next;
end loop;

```

14. Show what is written by the following segment of code:

```
List := new Node_Type'(Info => 5, Next => null);
Ptr := List;
for Count in 1..4 loop
 Ptr.all.Next := new Node_Type'(Info => Count, Next => List);
 Ptr := Ptr.all.Next;
end loop;
for Count in 1..6 loop
 Integer_IO.Put (Ptr.all.Info);
 Text_IO.New_Line;
 Ptr := Ptr.all.Next;
end loop;
```

15. Show what is written by the following segment of code:

```
List := new Node_Type'(Info => 5, Next => null);
Ptr := List;
for Count in 1..4 loop
 Ptr.all.Next := new Node_Type'(Info => Count, Next => List);
 Ptr := Ptr.all.Next;
end loop;
for Count in 1..6 loop
 Integer_IO.Put (Ptr.all.Info);
 Text_IO.New_Line;
 Ptr := Ptr.all.Next.all.Next;
end loop;
```

16. Show what is written by the following segment of code starting with the statement that follows the procedure:

```
procedure Attach (Value : in Natural;
 Ptr : in out Node_Ptr) is
begin
 Ptr := new New_Node;
 Ptr.all.Info := Value;
end Attach;
Attach (Value => 2, Ptr => List);
Attach (Value => 4, Ptr => List.all.Next);
Attach (Value => 6, Ptr => List.all.Next.all.Next);
Ptr := List;
loop
 exit when Ptr = null;
 Integer_IO.Put (Ptr.all.Info);
 Text_IO.New_Line;
 Ptr := Ptr.all.Next;
end loop;
```

17. Answer True or False, using the declarations below. If the answer is False, explain why. (The following declarations are global to the program.)

```
type Node_Type;
type Ptr_Type is access Node_Type;
```

```
type Node_Type is
record
 Info : Element_Type;
 Next : Ptr_Type
end record;
```



```
Ptr : Ptr_Type;
One_Node : Node_Type;
```

- (a) The space for the variable `Ptr` is dynamically allocated at run time.
- (b) The space for `Ptr.all` is dynamically allocated at run time.
- (c) `Ptr` is undefined until the new operator is used to assign it a value.
- (d) After the declarations, `Ptr = null`.
- (e) After the statement `Ptr := new Node_Type, Ptr.all.Next` is null.
- (f) Because `Ptr` accesses a record of `Node_Type`, `Ptr` and `One_Node` take the same amount of space in memory.
- (g) The declaration of `One_Node` is syntactically incorrect, because `Node_Type` records can only be allocated dynamically.

18. Discuss changes, if any, that would have to be made to the list operations if nonunique (duplicate) keys are allowed in the list.

19. The generic procedure `Traverse` of our list package raises the exception `KEY_ERROR` if the user's supplied function (formal generic procedure `Process`) changes the key of an element.

- (a) The comments given in our package declaration do not describe the state of the list when this exception is raised. Complete the documentation of this exception based on the code of the body of procedure `Traverse`.
- (b) Suggest how the body of procedure `Traverse` might be modified so that if the exception `KEY_ERROR` is raised, the list is unchanged.

20. The generic procedure `Traverse` of our list package has a generic formal procedure parameter called `Process`. Procedure `Process` has a single parameter, `Element`. Discuss the virtues of changing the mode of parameter `Element` from `in out` to `in`. What are the virtues of leaving it mode `in out`?

21. The list class is to be extended with a Boolean function, `Value_In_List`, which inputs a list and a `Key_Type` value and determines whether there is an element with this key in the list.

- (a) Write the specifications for this function.
- (b) Write the function using the sequential list implementation from the chapter.
- (c) Write the function using the linked list implementation from the chapter.
- (d) Describe `Value_In_List` in terms of Big-O.

22. The list ADT is to be extended with procedure `Merge_Lists`, with the following specification:

```
procedure Merge_Lists (List_1 : in out List_Type;
 List_2 : in out List_Type;
 New_List : out List_Type);
-- Merges the two lists List_1 and List_2 into a single list
-- Preconditions : New_List is empty
-- No key is present in both List_1 and List_2

-- Postconditions : List_1 and List_2 are empty
-- New_List contains all the elements
-- from List_1 and List_2
```

- (a) Write the body of `Merge_Lists` using the sequential list implementation described in the chapter.
- (b) Write the body of `Merge_Lists` using the linked list implementation described in the chapter.
- (c) Describe the algorithm you used for `Merge_Lists` in terms of Big-O.
- (d) Compare the implementation of `Merge_Lists` using sequential and linked implementations in terms of space requirements.

23. The list ADT is to be extended by the addition of procedure `Split_List`, which has the following specifications:

```

procedure Split_List (Main_List : in out List_Type;
 Split_Key : in Key_Type;
 List_1 : out List_Type;
 List_2 : out List_Type);
-- Divides a list into two lists according the key value of each
-- element. All elements with keys less than Split_Key will be put
-- into List_1, all elements with keys greater than or equal to
-- Split_Key will be put into List_2.

-- Preconditions : List_1 and List_2 are empty.

-- Postconditions : Main_List is empty.
-- List_1 contains all elements from Main_List with
-- keys less than Split_Key.
-- List_2 contains all elements from Main_List with
-- keys greater than or equal to Split_Key.

```

- (a) Write the body of `Split_List` using the sequential list implementation described in the chapter.
- (b) Write the body of `Split_List` using the linked list implementation described in the chapter.
- (c) Describe the algorithm you used for `Split_List` in terms of Big-O.
- (d) Compare the implementation of `Split_List` using sequential and linked implementations in terms of space requirements.

24. The list ADT is to be extended by the addition of procedure `Get_Element`, which inputs `List` and an integer, `Position`, and returns a copy of the element in the specified list position. For instance, if `Position` is 1, a copy of the first list element is returned; if `Position` is 12, a copy of the twelfth list element is returned. If `Position` is greater than the number of elements in the list, `CONSTRAINT_ERROR` is raised.

- (a) Write the specification for procedure `Get_Element`.
- (b) Write the body of `Get_Element` using the sequential list implementation described in the chapter.
- (c) Write the body of `Get_Element` using the linked list implementation described in the chapter.
- (d) Compare the sequential and linked versions of `Get_Element` in terms of Big-O.

25. The list ADT is to be extended by the addition of generic procedure `Reverse_Traverse`, which inputs `List` and calls the user-supplied procedure `Process` for each of the elements in reverse key order (from largest to smallest key). Unlike the procedure `Traverse` given in the list ADT in this chapter, procedure `Reverse_Traverse` may not change any element.

- (a) Write the specifications for `Reverse_Traverse`.
- (b) Write the body of `Reverse_Traverse` using the sequential list implementation described in the chapter.
- (c) What supporting data structure could be used to write `Reverse_Traverse` using the linked list implementation described in the chapter? Write the procedure body. (You may instantiate a package for the supporting ADT.)
- (d) Compare the sequential and linked versions of `Reverse_Traverse` in terms of code length, space requirements, and Big-O.

Use the following declarations in Exercises 26-29:

```

type Node_Type; -- Incomplete type declaration
type Node_Ptr is access Node_Type;

type Node_Type is
 record
 Info : Element_Type;
 Next : Node_Ptr;
 end record;

type List_Type is
 record
 Count : Natural := 0; -- Length of List
 First : Node_Ptr; -- Designates first node in list
 Current : Node_Ptr; -- Designates "current" node
 end record;

```

These list declarations support a list that is used for many tasks that require sequential processing; that is, the first element is processed, then the second, then the third, and so on. `First` always points to the first element in the list, whereas `Current` is used mark the "current" position in traversing the list.

26. `List` is an object of type `List_Type`. Draw a sketch of `List` when there are four nodes, and the third node is the current list position.

27. Procedure `Reset` inputs `List` and resets the `Current` field to the beginning of the list of elements.

(a) Write the specification of procedure `Reset`.

(b) Write the body of procedure `Reset` using the declarations above.

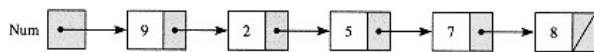
28. Procedure `Get_Current_Element` inputs `List` and returns the `Info` part of the "current" element in the list, as well as a Boolean flag indicating whether there are any more elements. The `Current` field of `List` should be advanced to point to the succeeding element. If there are no more elements, `No_More` should be set to `True` and `Current` should be reset to the beginning of the list.

(a) Write the specifications of procedure `Get_Current_Element`.

(b) Write body of `Get_Current_Element` using the declarations above.

29. Show how the declarations above would be modified to implement the list as an array-based sequential list.

30. Some computers have fairly small ranges of allowable integers. One way to store a really large integer is to put each digit in a node of a linked list. The following list represents the integer 92578:



Float numbers, of course, are allowed a much greater range than are integers, so the large integer can be converted into a variable of type `Float`. Write a function called `Make_Float` that inputs a pointer to a linked list of digits that represent a large integer, and returns the float-number equivalent of the number.

## Programming Problems

1. Using the `Address_Book` ADO developed in this chapter, write a program to implement the address book specification given on page 315.
2. Your assignment is to extend the List ADT by adding the following operations:

| <i>Name</i>                   | <i>Type</i>       | <i>Purpose</i>                                                                                                                                                                                                                                                                                                                                                          |
|-------------------------------|-------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Value_In_List</code>    | Function          | Determine whether there is an element with the specified Key value in List.                                                                                                                                                                                                                                                                                             |
| <code>Merge_Lists</code>      | Procedure         | Merge the elements in <code>List_1</code> and <code>List_2</code> into a single list, <code>New_List</code> .                                                                                                                                                                                                                                                           |
| <code>Split_List</code>       | Procedure         | Split the elements in <code>Main_List</code> into two lists, <code>List_1</code> and <code>List_2</code> , according to the key value of each element. All elements with keys less than <code>Split_Value</code> will be put into <code>List_1</code> ; all elements with keys greater than or equal to <code>Split_Value</code> will be put into <code>List_2</code> . |
| <code>Get_Element</code>      | Procedure         | Return a copy of the element in the <i>N</i> th position in the List.                                                                                                                                                                                                                                                                                                   |
| <code>Reverse_Traverse</code> | Generic Procedure | Traverse the List elements in reverse key order (from largest to smallest key) executing <code>Process</code> with each element.                                                                                                                                                                                                                                        |

The first part of your assignment is to write the specifications of each operation, in the following format:

### Operation (parameter list) Function:

**Preconditions:** \_\_\_\_\_

**Postconditions:** \_\_\_\_\_

**Exceptions:** \_\_\_\_\_

Once you have added these new operation specifications to Specification 8-1 (page 309), you should design and add their bodies to Body 8-3 implemented with linked lists.

### Testing the Operations

Write a test plan describing the test cases needed for each operation. Your grade will be determined in part by the completeness of your test plan.

Create a batch test driver that will execute these test cases. Write the test output to a file, as well as to the screen. Be sure to identify each test case with labels in your output. Print "before" and "after" values to show the results of each operation.

Your test driver may use any of the operations in the List ADT from this chapter.

**Turn In**

1. listings of your package specification and body
  2. a copy of your test plan
  3. program listing of the test driver
  4. hard copy of the file containing the test results (use a pen or pencil to mark each test as successful or not)
3. Your assignment is to track the corporate careers of some up-and-coming executives who are busily changing jobs, being promoted and demoted, and, of course, getting paid.

In this (admittedly unrealistic) version of the corporate world, people either belong to a company or are unemployed. The list of people the program must deal with is not fixed; initially there are none, and new people may be introduced by the JOIN command (see below).

Executives within a company are ordered according to a seniority system and are numbered from 1 to  $N$  (the number of people in the company) to indicate their rank: 1 is the lowest rank and  $N$  is the highest. A new employee always enters at the bottom of the ladder and hence always starts with a rank of 1. When a new person joins a company, the rank of everyone in the company increases by one, and when an employee quits, the rank of employees above him or her in that company decreases by one. Promotions can also occur and affect the ranks in the obvious way.

Naturally, salaries are based on rank. An employee's salary is Rank \* \$1000. Unemployed people draw \$50 in unemployment compensation.

**Input**

1. From file COMPANY: The company names are listed one per line. There will be at most 20 companies. Company names are at most 10 characters and do not contain embedded blanks.
2. From the keyboard: Commands are as listed below. Person and Company names are at most 10 characters and do not contain embedded blanks.

*JOIN* <person> <company>

<Person> joins the specified <company>. This may be the first reference to this person, or he or she may be unemployed. The person does not currently belong to another company. Remember that when a person joins a company he or she always starts at the bottom.

*QUIT* <person>

<Person> quits his or her job and becomes unemployed. You may assume that the person is currently employed.

*CHANGE* <person> <company>

<Person> quits his or her job and joins the specified new <company>. You may assume that the person is currently employed.

*PROMOTE* <person>

<Person> is moved up one step in the current company, ahead of his or her immediate superior. If the person has highest rank within the company, no change occurs.

*DEMOTE* <person>

<Person> is moved one step down in the current company, below his or her immediate subordinate. If the person has lowest rank within the company, no change occurs.

*PAYDAY*

Each person is paid his or her salary as specified above. (You must keep track of the amount each person has earned from the start of the program.)

**EMPLOYEES <company>**

The current list of employees should be printed for the specified <company>. The employees must be printed in order of rank; either top to bottom or bottom to top is appropriate.

**UNEMPLOY**

The list of unemployed people should be printed.

**DUMP**

Print the employees in each company, as specified under the EMPLOYEES command above, then print the unemployed people. Label the output appropriately.

**END**

Stop accepting commands.

Note that the CHANGE, PROMOTE, and DEMOTE commands do not tell you the person's current employer; you will have to search the data structure to find the person.

**Output (To Screen and to File EMPLOY.OUT)**

1. Echo print all commands, and print out a message that indicates what action has been taken. (For the EMPLOYEES and UNEMPLOY commands, print out the information specified in the Input section above.)
2. After all the commands have been processed, print out one list consisting of all the people who have been mentioned in any command and the total amount of money they have accumulated. The list should be sorted by decreasing order of total salary accumulated.

**List Class**

The list of employees for each company should be implemented using the linked list ADT in this chapter. If necessary, you may specify and implement additional operations for the List ADT.

**Testing**

You may want to test your program on the following sample data:

**File COMPANY:**

Borland

Microsoft

IBM

Digital

Compaq

NEC

XEROX

**Commands from keyboard:**

JOIN David XEROX

JOIN Mario XEROX

JOIN John Digital

JOIN Fred Digital

JOIN Phil IBM

CHANGE Fred NEC

JOIN Miriam Digital

JOIN Sharon Microsoft

JOIN Harvey Digital  
CHANGE Miriam Borland  
PAYDAY  
EMPLOYEES Digital  
JOIN Marge Borland  
JOIN Lesley Microsoft  
JOIN Sam Digital  
JOIN George NEC  
JOIN Bob Borland  
JOIN Susan IBM  
JOIN Joshua Digital  
JOIN Max NEC  
PAYDAY  
EMPLOYEES IBM  
EMPLOYEES Digital  
EMPLOYEES NEC  
JOIN Tim IBM  
DEMOTE Harvey  
PROMOTE Max  
DEMOTE Marge  
CHANGE Marge IBM  
QUIT John  
PAYDAY  
QUIT Mario  
QUIT David  
PROMOTE Marge  
PROMOTE Marge  
PAYDAY  
UNEMPLOY  
EMPLOYEES IBM  
EMPLOYEES XEROX  
JOIN John Compaq  
JOIN Ralph Compaq  
QUIT Phil  
JOIN Phil Compaq  
DUMP  
CHANGE Marge Compaq  
CHANGE Miriam Compaq  
CHANGE Fred Compaq  
CHANGE Susan Compaq  
QUIT Tim  
PAYDAY  
EMPLOYEES Compaq  
JOIN Mario XEROX  
JOIN David XEROX

EMPLOYEES XEROX  
JOIN Tim Compaq  
PROMOTE Tim  
PROMOTE Fred  
DEMOTE Miriam  
JOIN Laszlo Digital  
PROMOTE Laszlo  
CHANGE Joshua Compaq  
PAYDAY  
PROMOTE Sharon  
DEMOTE Lesley  
PROMOTE Bob  
DEMOTE Bob  
DEMOTE John  
PAYDAY  
DUMP  
END

***Turn In***

1. source program listing
2. list package specification and body
3. hard copy of file EMPLOY.OUT, containing the output of the test execution

*(This programming assignment was developed from an idea by Jim Bitner.)*



**9—****Lists Plus****GOALS**

After reading this chapter, you should be able to

- implement a linked list with a header node or a trailer node or both
- implement a circular linked list
- implement a doubly linked list
- instantiate more powerful list traversal operations
- discuss design issues for implementing lists with nonunique keys

In Chapter 8 you saw how a useful data structure, the sorted list, can be represented as either a sequential or a linked structure. There are many variations of list structures: lists with restricted access (such as the stack and the queue) and lists with special requirements whose operations can be simplified by using different implementations of the linked list.

As we consider many possible implementations, note that the specifications of the class *operations* do not change. From the application programmer's perspective, the list (or stack or queue) is a logical structure that can be manipulated through a set of operations. It shouldn't matter to the application program if we change the implementation. We repeat this point over and over because it may be the most important concept you will learn from this book. In this chapter we develop additional implementations of the sorted list class described by Specification 8-1. Each different implementation has advantages and disadvantages. No single implementation is best for all sorted list needs.

The generic list package developed in Chapter 8 contained a generic list traversal procedure that we used to instantiate procedures for processing all of the elements in a list. In this chapter we also show how to create more powerful list operations from this generic list traversal procedure.

### Linked Lists with Dummy Nodes

In writing the insert and delete algorithms for linked lists, we saw that special cases arise when we are dealing with the first node of the list. And in writing the search algorithm, we found that we needed three different exit conditions (found the key, passed by the spot where it should be, and reached the end of the list). One way to simplify all three of these algorithms is to make sure that we *never* insert or delete at the ends of the list. We can accomplish this simplification through the use of dummy nodes. A *dummy node* is a node in the linked list that does not contain list data. Dummy nodes simply act as placeholders.

#### Header Node

Placeholder node at the beginning of a list, use to simplify list processing.

#### Trailer Node

Placeholder node at the end of a list, used to simplify list processing. The key of a trailer node must be greater than any valid key for the list.

A **header node** is a dummy node placed at the beginning of the list. Like any node, it is a record with `Info` and `Next` fields. However, the `Info` field contains no list data; it contains logical garbage. A **trailer node** is a dummy node placed at the end of the list. The trailer node must contain an element whose key is greater than any valid key possible for the list. Figure 9-1(a) shows a list of three elements with a header node and a trailer node. Figure 9-1(b) shows an empty list with a header node and a trailer node. The keys in these lists are seven character names. The trailer nodes in Figure 9-1 contain the key "zzzzzzz"—a string that is greater than any valid name.

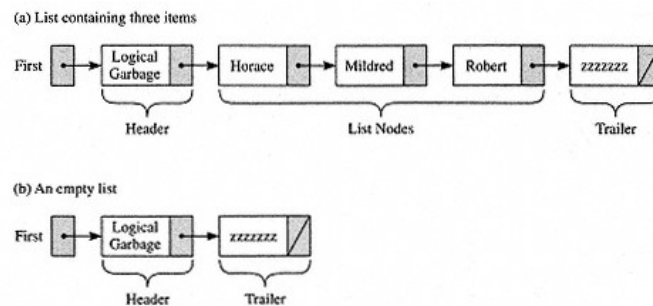


Figure 9-1  
A Linked List with Header and Trailer Nodes

As usual, a new list implementation requires us to change the private part of Specification 8-1. This time we need to change our list type so that every list object is initialized with a header node and a trailer node. Here is the private part with the necessary declarations:

```
private -- Linked List with Header and Trailer nodes

type Node_Type; -- Incomplete type declaration

type Node_Ptr is access Node_Type; -- Access to a node

type Node_Type is -- Complete type declaration
 record
 Info : Element_Type; -- One element
 Next : Node_Ptr; -- Link to next node in the list
 end record;

type List_Type is
 record -- Each list starts with a header and a trailer node
 First : Node_Ptr := new Node_Type'(Info => Max_Element,
 Next => new Node_Type'(
 Info => Max_Element,
 Next => null));
 end record;

end Sorted_List;
```

Rather than being a pointer, `List_Type` is a record type containing only an external pointer to the list. Why did we enclose the external list pointer in a record declaration? To answer this question we need to review how Ada initializes certain objects. There are only two types for which Ada provides initial values: access types and record types. When any access variable is elaborated, it is assigned an initial value of `null`. Ada's record type provides a slightly different form of initialization. A record type declaration may contain an initial value for each field. When a variable of a record type with initial values is elaborated, these initial values are assigned to the record variable's fields. If an initial value is not supplied for a field in the record type declaration, no initial value is assigned to that field when the record variable is elaborated.

The initialization of access variables to `null` is a fine way to initialize lists whose empty lists contain no nodes. But when we use header and trailer nodes, an empty list contains two nodes [Figure 9-1(b)]. In our declarations for a linked list with header and trailer nodes, we use a record type with an initial value. Whenever a list object's declaration is elaborated, the allocator `new` obtains memory for a header node and sets the external list pointer, `First`, to point to this new node. The `Info` field of the header node is assigned the value `Max_Element`. The `Next` field of the header node is set to the value given by a second invocation of the allocator `new` that obtains memory for a trailer node. The `Info` field of this trailer node is assigned the value `Max_Element` and its `Next` field is assigned `null`. Thus when an object of type `List_Type` is elaborated, memory is allocated for two nodes that are linked together [Figure 9-1(b)].

Where do we get `Max_Element`? Its key must be greater than any possible for the list. Only the application programmer instantiating a package from our generic sorted list knows what the keys and elements actually are. Therefore, the programmer must supply us with an element whose key is greater than any possible key in the application. How can our

package get `Max_Element` from the application programmer? We use a generic formal object. The application programmer supplies an actual value when he or she instantiates a sorted list package. Here is a portion of Specification 8-1 showing the additional generic parameter `Max_Element`:

```
generic -- Linked List with Header and Trailer nodes

type Element_Type is private; -- The type of element in the list
type Key_Type is limited private; -- The type of key

Max_Element : in Element_Type; -- An element with key greater
 -- than any legitimate element

-- The user must supply a function that returns the Key of an Element.
with function Key_Of (Element : in Element_Type) return Key_Type;
-- The user must supply functions for comparing Keys.
with function "=" (Left : in Key_Type; Right : in Key_Type)
 return Boolean;

with function "<" (Left : in Key_Type; Right : in Key_Type)
 return Boolean;

package Sorted_List is
.
.
-- List Operations as before (Specification 8-1)
.
.
private

 -- Private declarations for linked list with header and trailer nodes

end Sorted_List;
```

Let's look at a sample instantiation of a list with header and trailer nodes. Here are the declarations for a parts list that we used to illustrate list instantiation in Chapter 8:

```
-- Instantiate a string package for part names
package Part_Strings is new Bounded_Strings (Max_Length => 10);

-- Declare a synonym so we won't have to use such a long name
subtype Name_String is Part_Strings.Bounded_String;

-- Inventory record
type Part_Rec is
 record
 Name : Name_String;
 Amount : Natural := 0;
 end record;

function Name_Of (Part : in Part_Rec) return Name_String;
```

The list elements are records containing a part name (the key) and a quantity. Here are the declarations required to instantiate a sorted list package for part records implemented as a linked list with header and trailer nodes:

```
-- A key larger than any legitimate key
Big_Key : constant Name_String := Part_Strings.To_Unbounded("zzzzzzzzz");
-- A record containing the key larger than any legitimate key
Big_Rec : constant Part_Rec := (Name => Big_Key,
 Amount => 0);

-- A sorted list class (implemented with header & trailer nodes)
package Part_List is new Sorted_List (Element_Type => Part_Rec,
 Key_Type => Name_String,
 Max_Element => Big_Rec,
 Key_Of => Name_Of,
 "=" => Part_Strings.Equal,
 "<" => Part_Strings."<");
```

### ***The Empty Operation***

How do we know whether the list is empty? Our old standby Empty operation

```
return List = null;
```

won't work when an "empty" list contains two nodes [Figure 9-1(b)]. Our Empty operation needs to check if there are *exactly* two nodes in the list.

```
function Empty (List : in List_Type) return Boolean is
begin
 -- Are there exactly two nodes (a header and a trailer)?
 return List.First.all.Next.all.Next = null;
end Empty;
```

Notice how the use of `.all` allows us to distinguish quickly between a component in an ordinary record and a component referenced by an access value. `First` is a component of record object `List`, and `Next` is a component in a node designated by `First`. Using the shorthand notation we discussed in Chapter 6, we could rewrite the return statement as

```
return List.First.Next.Next = null;
```

This shorthand notation makes it difficult to determine whether `First` is a record with a field called `Next` or an access value that designates a record with a field called `Next`. The only way to resolve this question is to look through the declarations.

### ***Traversing a List with a Header and Trailer***

Traversing a list that has dummy nodes is not much different from traversing any other linear linked list, except for the fact that not all the nodes contain list elements. The user does not want the values of the header and trailer to be processed (they are *internal* to the list's implementation). We must set up our traverse loop to avoid these nodes. We use the same

algorithm we have developed for traversing a linked list without dummy nodes, but there are minor changes in the implementation.

### Traverse List

```

Location := start of list
loop
 exit when all elements have been processed
 Process (Location.all.Info);
 Location := Location.all.Next;
end loop

```

Two things must change to implement this algorithm for a list that has a header and trailer node: the initialization of the loop control variable (`Location`) and the terminating condition. Where is the "start of list" when the list has a header node? The first node in the list that contains user data is the second physical node (`List.First.all.Next`), so we initialize `Location` to point to this node. Where is the end of the list? The last node in the list that contains user data is the node before the trailer node. We do not want to process the information in the trailer node, so our terminating condition must stop the loop when `Location` designates the trailer node. How can we tell when `Location` is designating the trailer node? The trailer node is the only node in the list whose `Next` field contains null; we exit the loop when `Location.all.Next` equals null. What if the list is empty? The first node examined has a null value in its `Next` field and we terminate without entering the loop. The resulting procedure is shown below.

```

procedure Traverse (List : in out List_Type) is
-- For a linked list with header and trailer nodes
 Element : Element_Type; -- To ensure no key changes
 Location : Node_Ptr; -- Designates current node
begin
 Location := List.First.all.Next; -- Start after the header node
 loop
 -- Exit when Location designates the trailer node
 exit when Location.all.Next = null;
 Element := Location.all.Info; -- Make a local copy of the element
 Process (Element); -- Call user's process procedure
 -- Make sure the Key wasn't changed
 if Key_Of (Element) /= Key_Of (Location.all.Info) then
 raise KEY_ERROR;
 else -- Put possibly modified element back into list
 Location.all.Info := Element;
 end if;
 Location := Location.all.Next; -- Move to next node
 end loop;
end Traverse;

```

### Finding a List Element in a List with a Header and Trailer

In procedure `Search_Linked_List` of Body 8-3 we searched the list looking for the key using a pair of local pointers, `Current` and `Previous`. (Remember the inchworm?) We modify this approach slightly for a list with header and trailer nodes. In Body 8-3, we initialized `Current` to point to the first node in the list, and set `Previous` to null

[Figure 9-2(a)]. For a list with header and trailer nodes, we initialize `Current` to point to the first node, and `Previous` to point to its predecessor—the header node [Figure 9-2(b)]. Figure 9-2(c) illustrates the initialization of these two pointers when the list is empty. Here is the logic for initializing the two pointers for the search procedure:

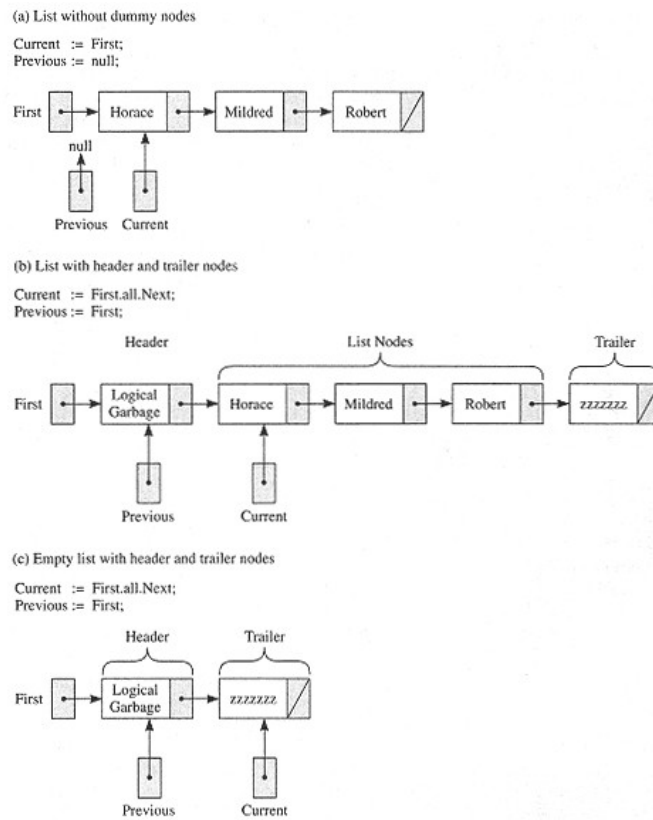


Figure 9-2  
Initialization for the Search

### Initialize for Search

```

Previous := First -- Previous designates header
Current := First.all.Next -- Current designates first "real" node

```

The search loop in procedure `Search_Linked_List` of Body 8-3 contained an exit statement with three conditions.

```

loop -- Each iteration, one node is checked
 exit when Current = null or else -- Reached the end
 Key < Key_Of (Current.all.Info) or else -- Passed by
 Key = Key_Of (Current.all.Info); -- Found it
 .
 .
 .
end loop;

```

Evaluating these three conditions every loop iteration can be time consuming. Because the trailer node's key is greater than any possible in the list, the search loop terminates either before or at the trailer node. Current never runs off the end of the list, so we no longer need to check for a `null` value. We loop until we come across a key that is greater than or equal to `Key`. Our search loop becomes

```

loop -- Each iteration, one node is checked
 exit when Key < Key_Of (Current.all.Info) or else -- Passed by
 Key = Key_Of (Current.all.Info); -- Found it
 .
 .
 .

```

Using header and trailer nodes simplifies the postconditions of our original search procedure by making it impossible for either `Pred_Loc` or `Location` to be returned as `null`. Specification 9-1 describes the local search procedure for a linked list with header and trailer nodes. Compare the postconditions of this specification to those of Specification 8-4.

### Specification 9-1—

#### Linked List Search, with Header and Trailer Nodes

```

procedure Search_Linked_List (List : in List_Type;
 Key : in Key_Type;
 Found : out Boolean;
 Pred_Loc : out Node_Ptr;
 Location : out Node_Ptr);
-- Searches for the location of Key in the List

-- Preconditions : The nodes in List are in ascending order.
-- List has Header and Trailer Nodes.
--
-- Postconditions : If Key is in List
-- Found is True.
-- Location designates the element with Key.
-- Pred_Loc designates the predecessor of the node
-- designated by Location.
-- else
-- Found is False.

```



```
--
-- Location designates the node that would follow a
-- node containing Key.
-- Pred_Loc designates the predecessor of the node
-- designated by Location.
```

Following the execution of the search procedure, if a matching key is found, `Location` points to the list node with that key and `Pred_Loc` points to its predecessor in the list [Figures 9-3(a) and 9-3(b)]. Note that if `Key` is the smallest key in the list, `Pred_Loc` points to its physical predecessor—the header node [Figure 9-3(b)]. If `Key` is not in the list, `Pred_Loc` and `Location` point to the nodes that would be before and after it if the key were in the list [Figures 9-3(c), 9-3(d), and 9-3(e)] even if the list is empty [Figure 9-3(f)].

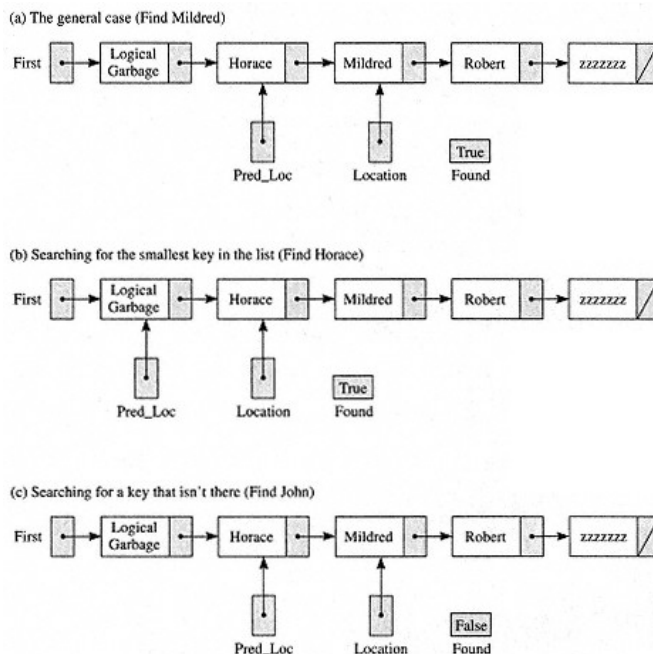
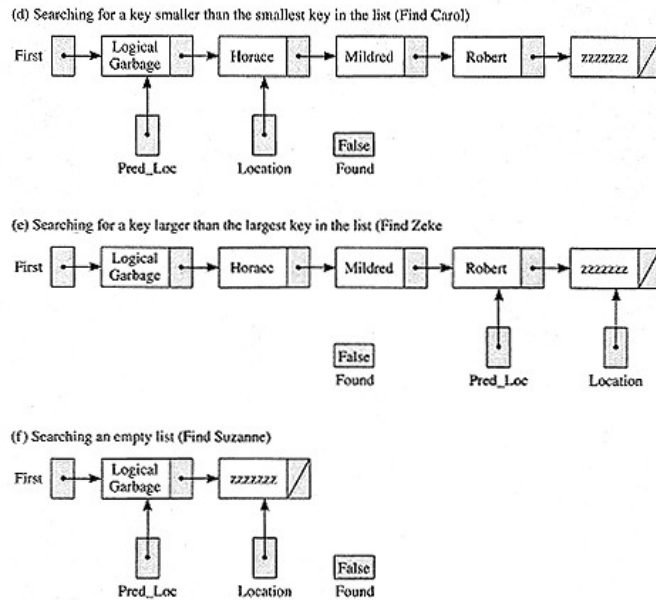


Figure 9-3  
Searching a List with Header and Trailer Nodes



The code to implement Specification 9-1 is shown below:

```

procedure Search_Linked_List (List : in List_Type;
 Key : in Key_Type;
 Found : out Boolean;
 Pred_Loc : out Node_Ptr;
 Location : out Node_Ptr) is
-- Search a linked list with header and trailer nodes
 Current : Node_Ptr; -- Local copy of Location
 Previous : Node_Ptr; -- Local copy of Pred_Loc
begin
 Current := List.First.all.Next; -- Start after the header node
 Previous := List.First; -- Predecessor is the header node
 loop --Each iteration, one node is checked
 exit when Key < Key_Of (Current.all.Info) or else -- Passed by
 Key = Key_Of (Current.all.Info); -- Found it
 Previous := Current; -- Move inchworm's tail
 Current := Current.all.Next; -- Move inchworm's head
 end loop;
 -- Copy results to out parameters
 Location := Current;
 Pred_Loc := Previous;
 Found := Key = Key_Of (Current.all.Info);
end Search_Linked_List;

```

```

procedure Insert (List : in out List_Type;
 Item : in Element_Type) is
-- For a linked list with header and trailer nodes
 Have_Duplicate : Boolean;
 Pred_Loc : Node_Ptr;
 Location : Node_Ptr;
begin
 Search_Linked_List (List => List,
 Key => Key_Of (Item),
 Found => Have_Duplicate,
 Pred_Loc => Pred_Loc,
 Location => Location);
 if Have_Duplicate then
 raise DUPLICATE_KEY;
 else -- Add between the nodes designated by Pred_Loc and Location
 Pred_Loc.all.Next := new Node_Type'(Info => Item,
 Next => Location);
 end if;
end insert;

```

```

exception
 when STORAGE_ERROR => -- Overflow when no storage available
 raise OVERFLOW;
end Insert;

procedure Delete (List : in out List_Type;
 Key : in Key_Type) is
-- For a linked list with header and trailer nodes
 Found : Boolean;
 Pred_Loc : Node_Ptr;
 Location : Node_Ptr;
begin
 Search_Linked_List (List => List,
 Key => Key,
 Found => Found,
 Pred_Loc => Pred_Loc,
 Location => Location);

 if not Found then
 raise KEY_ERROR;
 else -- Delete the node designated by Location
 Pred_Loc.all.Next := Location.all.Next; -- Unlink element
 Free (Location); -- Recycle node memory
 end if;
end Delete;

```

### ***Header, Trailer, Both, or Neither?***

Depending on the particular application, you may want to use a header, a trailer, both, or neither. While header and trailer nodes are nice because of the simpler search, insertion, and deletion algorithms, they do have drawbacks. Header nodes simplify the insert and delete procedures by eliminating the special case of changing the external list pointer. The disadvantages of using a header node include slightly more complicated declarations for `List_Type` and the extra memory used. If your application uses many lists, the amount of memory used by header nodes could become significant.

Trailer nodes simplify the search by eliminating one of the three exit conditions from our original search loop. The disadvantages of trailer nodes include slightly more complicated type declarations for `List_Type`, use of extra memory, and the selection of a key value that is greater than any possible in the list. A programmer using this version of our sorted list class must be very careful to make sure that the key of `Max_Element` he or she supplies as a generic parameter is truly "larger" than any valid key. (Maybe there is someone named "zzzzzz"!)

### **Circular Linked Lists**

The linked list data structures that we implemented in Chapter 8 and in the previous section are characterized by a *linear* (line-like) relationship between the elements: Each element (except the first one) has a unique predecessor, and each element (except the last one) has a unique successor. The algorithm we developed in Chapter 8 for inserting into a linear

linked list is  $O(1)$  for adding an element to the beginning of the list and  $O(N)$  for adding to the end of the list. These differences were confirmed by the insertion times found in our experiments.

---

#### Circular Linked List

A list in which every node has a successor; the "last" element is succeeded by the "first" element.

---

It is not uncommon for the data we want to add to a list to already be in order. Sometimes people manually sort raw data before turning it over to a data entry clerk. Data produced by other programs is often in some order. With sorted input data, we always insert at the end of the list. It is ironic that the work done previously to order the data now results in maximum insertion times. However, we can improve the efficiency of inserting at the end of a list by using a **circular linked list**. Circular lists are good for applications that require access to both ends of the list.

In a circular linked list, the pointer in the `Next` field of the last node points back to the first node instead of containing `null` (Figure 9-4). We can start at any node in the list and traverse the whole list. Thus we can make our external pointer to the list point to any node and still be able to access every node in the list. In fact, a truly circular list may not have a first or a last node—just a ring of elements linked to each other.

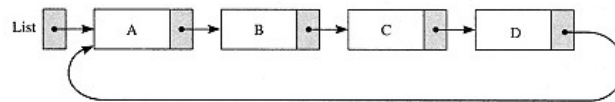


Figure 9-4  
A Circular Linked List

By letting the external pointer point to the (logical) last node in the list we can easily access both ends of the list: `List` points to the last node, and `List.all.Next` points to the first node in the list [Figure 9-5(a)]. The `Next` field of the node in a list with only one element points to the element itself [Figure 9-5(b)]. An empty circular list is represented by a `null` value for the external pointer to the list [Figure 9-5(c)].

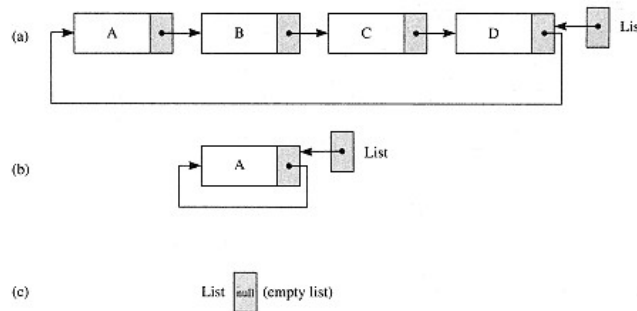


Figure 9-5  
Circular Linked Lists with the External Pointer Pointing to the List Rear Element

We mentioned this type of list structure in Chapter 7, when we discussed circular linked queues. We let the external pointer, `Queue`, point to the Rear node; `Queue.all.Next` then points to the Front node. Let's look at how to implement the `List` class (Specification 8-1) as a circular linked list. We develop some of the operations in detail; the rest are left as exercises.

There is no need to change any of the private part declarations of Specification 8-1 to make the list circular, rather than linear. After all, the fields in the nodes are the same—only the value of the next field of the last node has changed. How does the circular nature of the list change the implementation of the list operations? Because an empty circular list is one with a `null` pointer, the `Empty` operation is not changed at all. However, using a circular list requires an obvious change in algorithms that traverse the list (`Traverse`, `Search`, and `Clear`, for instance). We no longer stop when the traversing pointer becomes `null`. Unless the list is empty, the pointer never becomes `null`. Instead we must look for the external pointer itself as a stop sign.

### Traversing a Circular List

Let's write the body of the generic procedure `Traverse` that is described in Specification 8-1. We initialize a temporary pointer, `Location`, to the beginning of the list. Because `List`, the external pointer, designates the last element in the list, we must initialize `Location` to `List.all.Next`. We continue until `Location` comes full circle—when `Location = List`.

#### Traverse

```
Location := List.all.Next -- Start with "first" node
loop
 Process (Location.all.Info)
 exit when Location = List
 Location := Location.all.Next
end loop
```

Note the similarity between this initialization code and the initialization code we used with a list with a header node. In each of these implementations, the first node in the list has a predecessor. In the list with a header node, the predecessor is the header node. Here it is the last node in the list. We have used a midtest loop to delay checking if `Location = List` until after we have processed the element. If we had used a pretest loop

```
Location := List.all.Next -- Start with "first" node
loop
 exit when Location = List -- Pretest loop fails to process last element in the list
 Process (Location.all.Info)
 Location := Location.all.Next
end loop
```

we always stop before processing the last element. This problem is easy to spot in a list containing only one element—when `List` and `List.all.Next` are equal [see Figure 9-5(b)]. In this case, `Location` is initialized to `List` and we exit the loop without processing the only element in the list. Therefore we have used a midtest looping structure; now the check is made after processing the element.

Because this midtest loop always processes one element, we have a problem when the list is empty [Figure 9-5(c)]. The assignment of `List.all.Next` to `Location` raises `CONSTRAINT_ERROR` when `List` is `null`. We can resolve this problem by checking for this possibility ahead of time and avoiding the loop if the list is empty. Here is the Ada code for procedure `Traverse`. It includes logic to raise `KEY_ERROR` as described in

Specification 8-1 to ensure that the process procedure provided by the caller does not change any keys and destroy the order of the list.

```

procedure Traverse (List : in List_Type) is
 Element : Element_Type; -- Local variable ensures keys aren't changed
 Location : Node_Ptr; -- Designates current node
begin
 if List /= null then -- Does the list have at least one element?
 Location := List.all.Next; -- Start at beginning of list
 loop
 Element := Location.all.Info; -- Local copy of the element
 Process (Element); -- Call user's procedure
 -- Make sure the Key wasn't changed
 if Key_Of (Element) /= Key_Of (Location.all.Info) then
 raise KEY_ERROR;
 else -- Put possibly modified element back into list
 Location.all.Info := Element;
 end if;
 exit when Location = List; -- Midtest loop
 Location := Location.all.Next; -- Move to next node
 end loop;
 end if;
end Traverse;

```

### ***Finding a List Element in a Circular List***

The Retrieve, Modify, Insert, and Delete operations all require us to search the list. Again we write a local search procedure to simplify these other operations. Specification 9-2 describes the local search procedure for a circular linked list.

### **Specification 9-2— Circular Linked List Search**

```

procedure Search_Circular_List (List : in List_Type;
 Key : in Key_Type;
 Found : out Boolean;
 Pred_Loc : out Node_Ptr;
 Location : out Node_Ptr);
-- Searches for the location of Key in the List
-- Preconditions : The nodes in List are in ascending order.
-- List is circular.
--
-- Postconditions : if List is empty
-- Found is False.
-- Location is null.
-- Pred_Loc is null.
-- elsif Key is in List
-- Found is True.
-- Location designates the element with Key.
-- Pred_Loc designates the predecessor of the node
-- designated by Location.

```

```
--
-- else
-- Found is False.
-- Location designates the node that would follow a
-- node containing Key.
-- Pred_Loc designates the predecessor of the node
-- designated by Location.
```

The *in* parameters for this procedure are an external pointer to the list and a key value to search for. The *out* parameters are a Boolean telling whether or not the key was found in the list, and two pointers to locate it.

Following the execution of the search procedure, if a matching key is found, *Location* points to the list node with that key and *Pred\_Loc* points to its predecessor in the list [Figure 9-6(a)]. Note that if *Key* is the smallest key in the list, *Pred\_Loc* points to its physical predecessor—the last node in the circular list [Figure 9-6(b)]. If *Key* is not in the list, *Pred\_Loc* and *Location* point to the nodes that would be before and after it if the key were in the list [Figure 9-6(c) and 9-6(d)]. If the list is empty, both pointers are returned as *null* [Figure 9-6(e)].

As before we search the linked list looking for the key using a pair of local access values, *Current* and *Previous*. We must modify the initialization of these pointers and loop termination conditions for the circular list. In the list version with header nodes, we initialized *Current* to point to the first node in the list and set *Previous* to its predecessor—the header node [Figure 9-2(b)]. For the circular list search, we again initialize

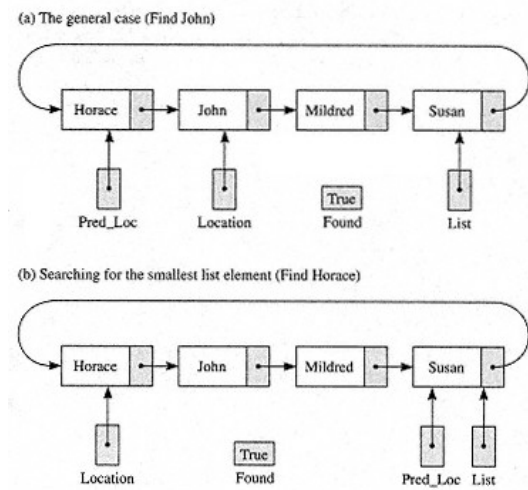
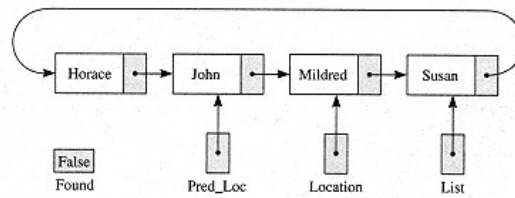


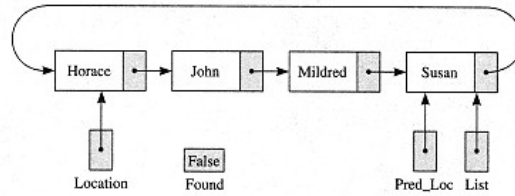
Figure 9-6  
Searching a Circular Linked List



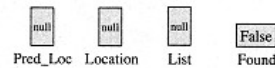
(c) Searching for element that isn't there (Find Kelly)



(d) Searching for element bigger than any in the list (Find Tom)



(e) Searching an empty list (Find Mildred)



Current to point to the first node, and Previous to point to its predecessor. But this time the predecessor is the last node in the list (Figure 9-7).

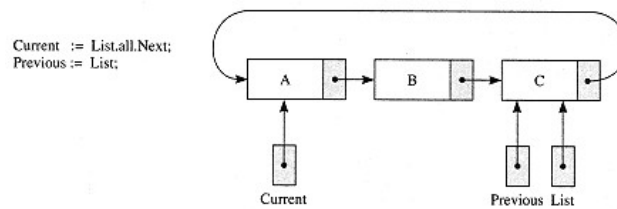


Figure 9-7  
Initializing for the Search of a Circular List

But wait a minute! Using this logic cannot give us the  $O(1)$  insertion at the end of the list that we wanted; it always start the search at the beginning of the list. To optimize insertions at the end of the list we need to add a test before our loop to see if the key should go at the end of the list. And like the traversal operation we developed earlier, to prevent a `CONSTRAINT_ERROR` we must check for an empty list before we begin our search loop. Here is the resulting search procedure:

```

procedure Search_Circular_List (List : in List_Type;
 Key : in Key_Type;
 Found : out Boolean;
 Pred_Loc : out Node_Ptr;
 Location : out Node_Ptr) is
 Current : Node_Ptr; -- Local copy of Location
 Previous : Node_Ptr; -- Local copy of Pred_Loc
begin
 if List = null then -- Is the list empty
 Found := False;
 Location := null;
 Pred_Loc := null;
 elsif Key_Of (List.all.Info) < Key then -- Is Key greater than
 Found := False; -- any key in the list?
 Location := List.all.Next; -- Key goes between last and first node
 Pred_Loc := List;
 else -- Search the list from the beginning
 Current := List.all.Next; -- Start at the beginning of
 Previous := List; -- the linked list
 loop -- Each iteration, one node is checked
 exit when Key < Key_Of (Current.all.Info) or -- Passed by
 Key = Key_Of (Current.all.Info); -- Found it
 Previous := Current; -- Move inchworm's tail
 Current := Current.all.Next; -- Move inchworm's head
 end loop;
 -- Copy results to out parameters
 Location := Current;
 Pred_Loc := Previous;
 Found := Key = Key_Of (Current.all.Info);
 end if;
end Search_Circular_List;

```

The loop in our search procedure executes until we find the desired key or we have passed the spot in the list where the key should be. Why don't we need the third condition (we reach the "end" of the list) that we had in our original search (Chapter 8) of a linked list? The check to see if the `Key` comes after the last node prior to the loop in the circular list search ensures that we never go beyond the end of the list.

### ***Inserting into a Circular List***

The algorithm to insert an element into a sorted circular linked list is similar to that for the linear list insertion:

#### **Insert**

```

Find the place where the new element belongs
Create space for the new element
Link the new element into the list

```

The first task is simple: we just call `Search_Circular_List`. If the search procedure finds an element with the same key of the element we wish to insert, the insert procedure

should raise `DUPLICATE_KEY`. The second task, creating space for the new element, is the same: we use the allocator `new`.

Now we are ready to link the new element into the list. This node is linked in after the node designated by `Pred_Loc`. If the list is empty, the search procedure returns `null` for `Pred_Loc` [Figure 9-6(e)]; there is no predecessor to link the new node after. In this case we want to make `List` point to the new node [Figure 9-8(a)]. Here is the algorithm:

#### Put Element into Empty List

```
List := new Node_Type; -- Get space
List.all.Info := Item; -- Assign element
List.all.Next := List; -- Link "last" to "first"
```

Once the list is no longer empty, the algorithm for *linking* an element into the list is the same at the beginning, the middle, and the end of the list; we link it between the nodes designated by `Pred_Loc` and `Location`. Figure 9-8 illustrates this algorithm for inserting into the "middle" [Figure 9-8(b)], the "beginning" [Figure 9-8(c)], and the "end" [Figure 9-8(d)]. In all three cases, we set the `Next` field of the new node to point to the same node that `Location` points, and change the `Next` field of the node designated by `Pred_Loc` so that it points to our new node. Figure 9-8(d) indicates that there is one more pointer to change when we add a new node to the end of the list. The external pointer, `List`, now must designate the new "last" node. This special case occurs when the key we wish to add is greater than any other key in the list. Here is the algorithm for adding a node to a nonempty list:

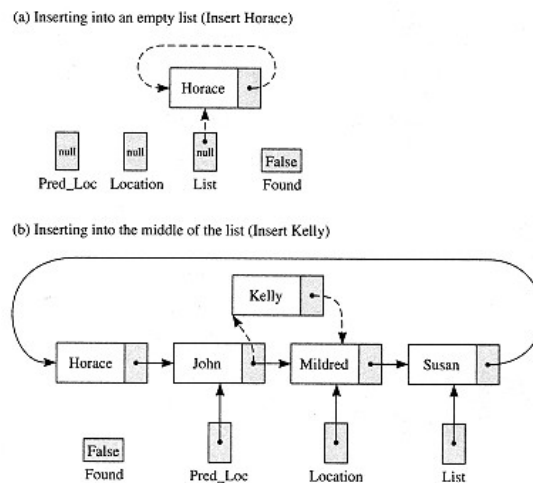
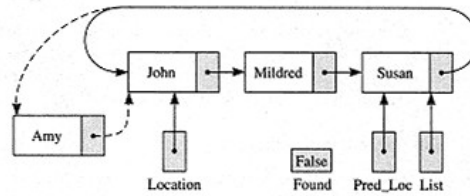
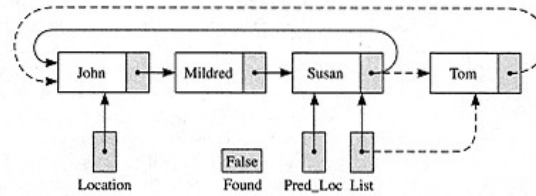


Figure 9-8  
Inserting into a Circular Linked List. Dashed arrows show the new values of pointers after the insert operation is complete.

(c) Inserting at the beginning of the list (Insert Army)



(d) Inserting at the end of the list (Insert Tom)



### Put Element into Nonempty List

```
Pred_Loc.all.Next := new Node_Type'(Info => Item, Next => Location);
if Key > Key of the last node in the list then
 List := List.all.Next;
end if;
```

The procedure body with exceptions added is shown below:

[illegible]

```

-- Did we add at end of list?
if Key_Of (Location.all.Info) < Key_Of (Item) then
 List := List.all.Next; -- Readjust external pointer
end if;
end if;
exception
 when STORAGE_ERROR => -- Overflow when no storage available
 raise OVERFLOW;
end Insert;

```

### Deleting from a Circular List

To delete an element from the circular linked list, we use the same general algorithm that we developed in Chapter 8:

#### Delete

```

Find the element in the list
Remove the element from the list
Recycle the memory used by the node

```

For the first task, we use our trusty helper, `Search_Circular_List`. On return from a call to this search procedure, `Location` points to the node we wish to delete, and `Pred_Loc` points to its predecessor in the list. To remove the node from the list, we reset its predecessor's `Next` field to jump over the node we are deleting:

```
Pred_Loc.all.Next := Location.all.Next;
```

That assignment statement works for at least the case of deleting from the middle of a list [Figure 9-9(a)]. What kind of special cases do we have to consider? In the linear list version, we had to check for deleting the first (or first-and-only) element. From our experience with the insertion operation, we might surmise that deleting the smallest element (the first node) of the circular list is *not* a special case; Figure 9-9(b) shows that guess to be correct. However, deleting the *only* node in a circular list is a special case, as we see in Figure 9-9(c). The external pointer `List` must be set to `null` to indicate that the list is now empty. We can detect this situation by checking to see if `Pred_Loc = Location` after the search; if so, the node we are deleting is the only one in the list.

(a) Deleting from the middle of the list (Delete John)

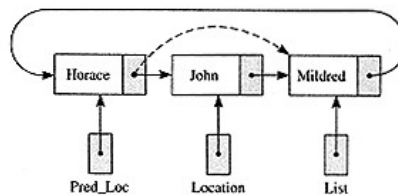
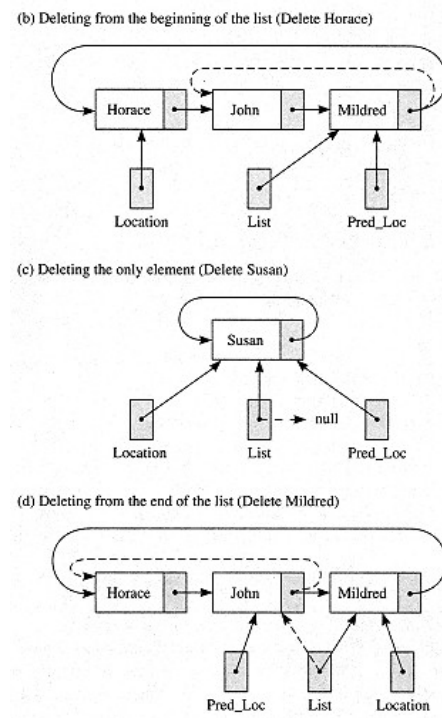


Figure 9-9  
Deleting from a Circular Linked List. Dashed arrows show the new values of pointers after the delete operation is complete.



We also might guess that deleting the largest list element (the last node) from a circular list is a special case. As Figure 9-9(d) illustrates, when we delete the last node, we first perform the general-case processing to unlink the node designated by `Location` from the list, then we reset the external pointer to point to its predecessor, which is designated by `Pred_Loc`. We can detect this situation by checking whether `Location = List` after the search.

The complete procedure body for `Delete` with exceptions added is shown below:

```

procedure Delete (List : in out List_Type;
 Key : in Key_Type) is
-- For a circular linked list
 Found : Boolean; -- True if Key exists in list
 Location : Node_Ptr; -- Designates the node containing Key
 Pred_Loc : Node_Ptr; -- Designates the node prior to that
 -- containing Key

```

```

begin
 Search_Circular_List (List => List,
 Key => Key,
 Found => Found,
 Pred_Loc => Pred_Loc,
 Location => Location);

 if not Found then
 raise KEY_ERROR;
 elsif Pred_Loc = Location then -- Deleting the only node in the list?
 Free (List); -- Recycle the only node in the list
 else
 Pred_Loc.all.Next := Location.all.Next; -- Unlink node
 if Location = List then -- Deleting last element?
 List := Pred_Loc;
 end if;
 Free (Location);
 end if;
end Delete;

```

Having worked through a number of the list operations in detail, we leave the implementation of the other operations specified for the List class as exercises.

### Doubly Linked Lists

We have discussed using circular linked lists to enable us to reach any node in the list from any starting point. Although this structure has advantages over a simple linear linked list, it is still too limited for certain types of applications. Suppose we want to be able to delete a particular node in a list, given only a pointer to that node (*Location*). This task involves changing the *Next* field of the node preceding the node designated by *Location*. As we saw in the previous chapter, however, given a pointer to a node, we cannot easily access its predecessor in the list.

Another task that is complex to perform (until you learn about recursion in Chapter 11) on a linear linked list is traversing the list in reverse. For instance, suppose we have a list of student records, ordered by grade point average (GPA) from lowest to highest. The Dean of Students might want a printout of the students' records, ordered from highest to lowest, to use in preparing the Dean's List.

---

#### Doubly Linked List

A linked list in which each node is linked to both its successor and its predecessor.

---

In cases like these, where we need to be able to access the node that *precedes* a given node, a **doubly linked list** is useful. In a doubly linked list, the nodes are linked in both directions. As shown in Figure 9-10, each node of a doubly linked list contains three parts:

- Info*: the data stored in the node
- Next*: the pointer to the following node
- Back*: the pointer to the preceding node

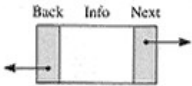


Figure 9-10  
A Node for a Doubly  
Linked List

A linear doubly linked list is pictured in Figure 9-11. Note that the `Back` field of the first node as well as the `Next` field of the last node contains a null value. The following declarations can be used in Specification 8-1 to declare such a list:

```
private -- Doubly Linked List

type Node_Type; -- Incomplete type declaration

type List_Type is access Node_Type; -- Access to a node
subtype Node_Ptr is List_Type; -- A synonym for our access type

type Node_Type is -- Complete type declaration
 record
 Info : Element_Type; -- One element
 Next : Node_Ptr; -- Link to next node in the list
 Back : Node_Ptr; -- Link to previous node in the list
 end record;

end Sorted_List;
```

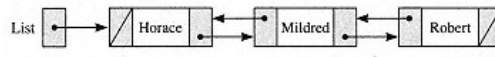


Figure 9-11  
A Linear Doubly Linked List

### Finding a List Element in a Doubly Linked List

In our search procedure, we no longer need to use the inchworm search to keep track of the current node's predecessor; instead we can get the predecessor to any node through its `Back` pointer as shown here:

```
Pred_Loc := Location.all.Back; -- Determine Location's predecessor
```

When searching a linear list like the one shown in Figure 9-11 for a key greater than any other in the list, our search loop terminates when `Location` becomes null. In this case `Location` does not designate a node and we cannot use the assignment statement `Pred_Loc := Location.all.Back` to determine a predecessor. To prevent `Location` from running off the end of the list, doubly linked lists are often implemented as either circular or with dummy nodes. Figure 9-12 shows a circular doubly linked list. If we do not make our doubly linked list either circular or use dummy nodes, we must use the inchworm search with `Pred_Loc` trailing along behind `Location` to search our list.

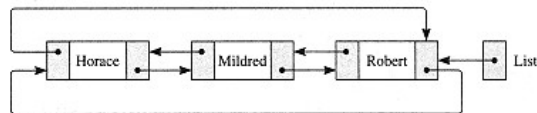


Figure 9-12  
A Circular Doubly Linked List. List points to the last node of the list.

Using a doubly linked circular list allows us to change the parameters of our local search procedure slightly. Specification 9-3 describes the local search procedure for a



circular doubly linked list. Because we no longer need `Pred_Loc`, we return one access value, `Location`. If `Key` is found in the list, `Location` points to the node containing it; otherwise, `Location` points to the node that is the *successor* in the list. (We can easily get to the preceding node through the `Back` pointer.)

### Specification 9-3—

#### Circular Doubly Linked List Search

```

procedure Search_Circular_List (List : in List_Type;
 Key : in Key_Type;
 Found : out Boolean;
 Location : out Node_Ptr);
-- Searches for the location of Key in the List

-- Preconditions : The nodes in List are in ascending order.
-- The List is circular doubly linked (List designates the last
-- node).
--
-- Postconditions : If List is empty
-- Found is False
-- Location is null
-- elsif Key is in List
-- Found is True
-- Location designates the element with Key
-- else
-- Found is False
-- Location designates the node that would follow a
-- node containing Key in the circular list.

```

The procedure shown below searches a circular doubly linked list; it implements Specification 9-3. The algorithm for this procedure is nearly identical to the algorithm we developed for the circular singly linked list in the last section:

```

procedure Search_Circular_List (List : in List_Type;
 Key : in Key_Type;
 Found : out Boolean;
 Location : out Node_Ptr) is
-- Search a circular doubly linked list
 Current : Node_Ptr; -- Local copy of Location

begin
 if List = null then -- Is the list empty
 Found := False;
 Location := null;
 elsif Key_Of (List.all.Info) < Key then -- Is Key greater than
 Found := False; -- any key in the list?
 Location := List.all.Next; -- Key goes between last and first node

```

```

else -- Search the list from the beginning
 Current := List.all.Next; -- Start at the beginning of the list
 loop -- Each iteration, one node is checked
 exit when Key < Key_Of (Current.all.Info) or -- Passed by
 Key = Key_Of (Current.all.Info); -- Found it
 Current := Current.all.Next;
 end loop;
 -- Copy results to out parameters
 Location := Current;
 Found := Key = Key_Of (Current.all.Info);
end if;
end Search_Circular_List;

```

### Operations on a Doubly Linked List

The algorithms for the insertion and deletion operations on a doubly linked list are somewhat more complicated than those for operations on a singly linked list. The reason is clear: There are more pointers to keep track of in a doubly linked list.

For example, consider the Insert operation. To link the new node, after a given node, in a singly linked list, we need to change two pointers [see Figure 9-13(a)]. The same operation on a doubly linked list requires four pointer changes [see Figure 9-13(b)].

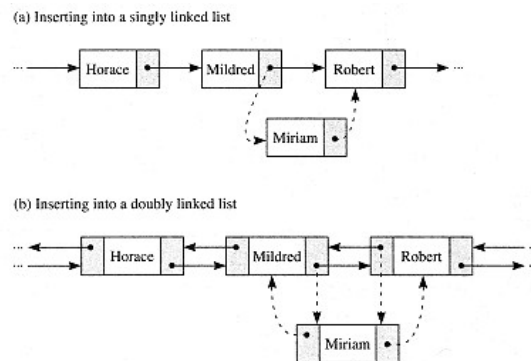


Figure 9-13  
Insertions into Singly and Doubly Linked Lists. Dashed arrows show the new values of pointers after the insert operation is complete.

To find the insertion place we call `Search_Circular_List`. On return, `Location` is `null` (if the list is empty) or designates the node that should *follow* the new node.

As usual, we allocate space for the new element using the `new` operator, and put the new element into the space. Now we are ready to link the new node into the list. Because of the complexity of the operation, it is important to be careful about the order in which you change the pointers. For instance, when inserting our new node before the node designated by `Location`, if we first change `Location.all.Back`, we lose our pointer to `Location`'s predecessor. The correct order for the pointer changes is illustrated in Figure 9-14. These four pointer changes are accomplished with the following assignment statements:

```

New_One := new Node_Type'(Info => Item,
 Back => Location.all.Back, ①
 Next => Location); ②

Location.all.Back.all.Next := New_One; ③

Location.all.Back := New_One; ④

```

The complete Insert procedure for a circular doubly linked list is shown below:

```

procedure Insert (List : in out List_Type;
 Item : in Element_Type) is
 -- For a circular doubly linked list
 Have_Duplicate : Boolean;
 Location : Node_Ptr; -- Designates successor of Item
 New_One : Node_Ptr; -- Designates node with Item
begin
 Search_Circular_List (List => List,
 Key => Key_Of (Item),
 Found => Have_Duplicate,
 Location => Location);

 if Have_Duplicate then
 raise DUPLICATE_KEY;
 elsif Location = null then -- Is the list empty?
 List := new Node_Type; -- Add first node to list
 List.all := (Info => Item, Next => List, Back => List);
 else
 New_One := new Node_Type'(Info => Item, -- Set up new node
 Next => Location,
 Back => Location.all.Back);
 Location.all.Back.all.Next := New_One; -- Update predecessor's Next
 Location.all.Back := New_One; -- Update successor's Back
 -- Did we add at end of list?
 if Key_Of (Location.all.Info) < Key_Of (Item) then
 List := New_One;
 end if;
 end if;
exception
 when STORAGE_ERROR => -- Overflow when no storage available
 raise OVERFLOW;
end Insert;

```

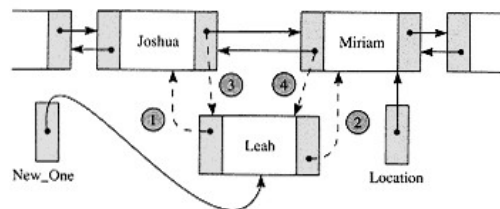


Figure 9-14  
Inserting a New Node into a Doubly Linked List

One of the useful features of a doubly linked list is that we don't need a pointer to a node's predecessor in order to delete the node. Through the `Back` pointer field, we can alter the `Next` field of the preceding node to make it jump over the unwanted node:

```
Location.all.Back.all.Next := Location.all.Next;
```

Then we make the `Back` pointer of the succeeding node point to the preceding node:

```
Location.all.Next.all.Back := Location.all.Back
```

Below is the version of procedure `Delete` for a circular doubly linked list. This operation is pictured in Figure 9-15.

```
procedure Delete (List : in out List_Type;
 Key : in Key_Type) is
 -- For a circular doubly linked list
 Found : Boolean;
 Location : Node_Ptr;
begin
 Search_Circular_List (List => List,
 Key => Key,
 Found => Found,
 Location => Location);

 if not Found then
 raise KEY_ERROR;
 elsif Location = Location.all.Next then -- Deleting the only node?
 Free (List); -- Recycle the only node in the list
 else
 Location.all.Back.all.Next := Location.all.Next; -- Unlink the
 Location.all.Next.all.Back := Location.all.Back; -- node
 if Location = List then -- Deleting last element?
 List := Location.all.Back;
 end if;
 Free (Location);
 end if;
end Delete;
```

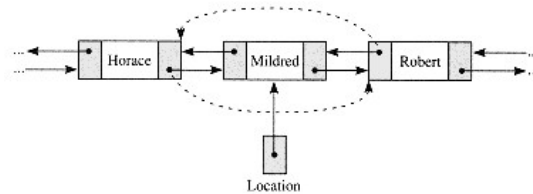


Figure 9-15  
Deleting from a Doubly Linked List. Dashed arrows show the new values of pointers after the Delete operation is complete.

### Doubly Linked Lists with Dummy Nodes

As we saw above, doubly linked lists can be circular. By now you probably realize that we can apply any of the techniques we developed for singly linked lists to doubly linked lists. In this section we look at two ways of using dummy nodes in a doubly linked list.

A doubly linked list may use headers and trailers as pictured in Figure 9-16. Because there are two dummy nodes to create and so many pointers to initialize to non-null values, using both header and trailer nodes in a doubly linked list requires a more complicated initialization scheme than can be accomplished with default initial field values in the `List_Type` record. We must write a `Create` operation that must be explicitly called by the application programmer to initialize any list objects. Every other list operation must contain the following precondition:

```
-- Preconditions : The List was originally initialized by procedure Create
```

This precondition puts the responsibility of initializing the list on the application programmer. If the application programmer does not initialize a list through a call to procedure `Create`, any other operation (`Insert`, `Empty`, `Retrieve`, and so on) they use on that list will fail. Here is the code for the `Create` operation:

```
procedure Create (List : in out List_Type) is
-- This procedure initializes a doubly linked
-- list with header and trailer nodes.

-- Preconditions : Create was NEVER previously called for this List.
--
-- Postconditions : An empty List is created.

begin
 -- Create the header and trailer nodes; link header to trailer
 List := new Node_Type'(Info => Max_Element,
 Next => new Node_Type'(Info => Max_Element,
 Next => null,
 Back => null),
 Back => null);
 -- Link the trailer back to the header
 List.all.Next.all.Back := List;
end Create;
```

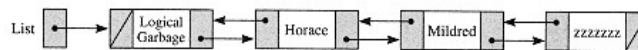


Figure 9-16  
A Doubly Linked List with Header and Trailer Nodes

The algorithms to retrieve, modify, insert, and delete in this list are nearly identical to those we have developed for the circular doubly linked list. The search and traverse operations require different initialization and loop exit conditions.

Figure 9-17 illustrates another form of the doubly linked list with dummy nodes. This one combines a circular linked list and a single dummy node. The single dummy acts as *both* a header node and a trailer node. Remember that the information in a header node is never accessed (it is considered logical garbage) and the key of the trailer node must be

greater than any valid key in the list. The algorithms to retrieve, modify, insert, and delete in this list are nearly identical to those we have developed for the circular doubly linked list. The search and traverse operations require different initialization and loop exit conditions. We start our search with the node after the header node and end it when we find the desired key or pass the spot where that key should be. Because the key of the trailer node is greater than any valid key in the list, we never go beyond the trailer node in our search. We leave the problem of initialization of this structure as an exercise.

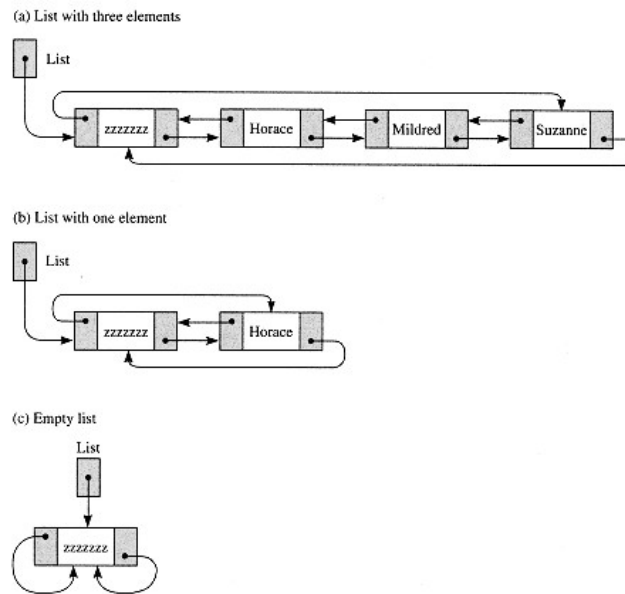


Figure 9-17  
A Doubly Linked List with a Single Dummy Node

## Comparison of Implementations

The following table shows the sizes (number of lines of source code) of the four different implementations we have developed:

|          | Comparison of Source Lines of Code |                             |                                             |                               |
|----------|------------------------------------|-----------------------------|---------------------------------------------|-------------------------------|
|          | <i>Linked List<br/>(Chapter 8)</i> | <i>Circular Linked List</i> | <i>Linked with Header and Trailer Nodes</i> | <i>Circular Doubly Linked</i> |
| Clear    | 10                                 | 15                          | 11                                          | 15                            |
| Empty    | 4                                  | 4                           | 4                                           | 4                             |
| Full     | 10                                 | 10                          | 10                                          | 10                            |
| Length   | 13                                 | 15                          | 13                                          | 15                            |
| Search   | 21                                 | 30                          | 20                                          | 23                            |
| Insert   | 22                                 | 26                          | 20                                          | 29                            |
| Delete   | 20                                 | 23                          | 18                                          | 22                            |
| Modify   | 17                                 | 17                          | 17                                          | 15                            |
| Retrieve | 18                                 | 18                          | 18                                          | 16                            |
| Traverse | 17                                 | 19                          | 17                                          | 19                            |
| Total    | 152                                | 177                         | 148                                         | 168                           |

Only the version using header and trailer nodes is shorter than the basic implementation developed in Chapter 8.

Is the efficiency of the insertion and deletion operations (in terms of Big-O) different in these variations? The following table compares them:

|           | Big-O Comparisons of Insert and Operations |                             |                                             |                               |
|-----------|--------------------------------------------|-----------------------------|---------------------------------------------|-------------------------------|
|           | <i>Linked List<br/>(Chapter 8)</i>         | <i>Circular Linked List</i> | <i>Linked with Header and Trailer Nodes</i> | <i>Circular Doubly Linked</i> |
| Insert    |                                            |                             |                                             |                               |
| Random    | $O(N)$                                     | $O(N)$                      | $O(N)$                                      | $O(N)$                        |
| Beginning | $O(1)$                                     | $O(1)$                      | $O(1)$                                      | $O(1)$                        |
| End       | <b><math>O(N)</math></b>                   | <b><math>O(1)</math></b>    | <b><math>O(N)</math></b>                    | <b><math>O(1)</math></b>      |
| Delete    |                                            |                             |                                             |                               |
| Random    | $O(N)$                                     | $O(N)$                      | $O(N)$                                      | $O(N)$                        |
| Beginning | $O(1)$                                     | $O(1)$                      | $O(1)$                                      | $O(1)$                        |
| End       | $O(N)$                                     | $O(N)$                      | $O(N)$                                      | $O(N)$                        |

The improvement here is for the circular lists where inserting at the end is  $O(1)$  rather than  $O(N)$ . Let's see what our experiments (described in Chapter 8) can tell us about these different implementations. Each number in the following table is the number of seconds required to insert or delete 1000 elements:

**Results of the Timed Implementation Comparisons (in seconds)**

|                           | <i>Linked<br/>List<br/>(Chapter 8)</i> | <i>Circular<br/>Linked<br/>List</i> | <i>Linked with<br/>Header and<br/>Trailer Nodes</i> | <i>Circular<br/>Doubly<br/>Linked</i> |
|---------------------------|----------------------------------------|-------------------------------------|-----------------------------------------------------|---------------------------------------|
| <b>Case 1: Random</b>     |                                        |                                     |                                                     |                                       |
| Insert                    | 0.71                                   | 0.75                                | 0.73                                                | 0.75                                  |
| Delete                    | 0.67                                   | 0.70                                | 0.69                                                | 0.73                                  |
| <b>Case 2: List End</b>   |                                        |                                     |                                                     |                                       |
| Insert                    | 1.27                                   | 0.03                                | 1.31                                                | 0.03                                  |
| Delete                    | 1.28                                   | 1.39                                | 1.31                                                | 1.32                                  |
| <b>Case 3: List Start</b> |                                        |                                     |                                                     |                                       |
| Insert                    | 0.03                                   | 0.04                                | 0.03                                                | 0.03                                  |
| Delete                    | 0.02                                   | 0.02                                | 0.01                                                | 0.02                                  |

These results show that for the specific computer on which the tests were run, there is not much difference between any of the dynamic memory implementations. Insertions at the end of circularly linked lists are the exception—a result we could predict from the Big-O determinations.

**Advanced Iterators**

Specification 8-1 contains the generic procedure `Traverse`. As we illustrated in Chapter 8, this generic procedure can be used to create customized procedures to process all of the elements in a sorted list. However, there are problems for which we must process every element in the list that cannot be solved with a simple instantiation of `Traverse`. Let's look at an example. The following code fragment uses Specification 8-1 to implement a sorted list of auto parts:

```
-- Instantiate a string package for auto part names
package Part_Strings is new Bounded_Strings (Max_Length => 10);

-- Declare a synonym so we won't have to use such a long name
subtype Name_String is Part_Strings.Bounded_String;

-- Inventory record
type Part_Rec is
 record
 Name : Name_String;
 Amount : Natural := 0;
 end record;

-- Auto Part List Package
package Auto_Part_List is new
 Sorted_List (Element_Type => Part_Rec,
 Key_Type => Name_String,
 Key_Of => Name_Of,
 "=" => Part_Strings.Equal,
 "<" => Part_Strings."<");
```



```
-- Declare a synonym so we won't have to use such a long name
subtype Part_List is Auto_Part_List.List_Type;
```

The following code fragment creates an iterator that displays the names of all parts for which there are fewer than 10 in stock:

```
procedure Check_One_Part (Part : in out Part_Rec) is
begin
 if Part.Amount < 10 then
 Part_Strings.Put (Item => Part.Name);
 Text_IO.New_Line;
 end if;
end Check_One_Part;

procedure Check_Inventory is new
 Auto_Part_List.Traverse (Process => Check_One_Part);
```

Like all of the iterators we instantiated in Chapter 8, `Check_Inventory` processes elements in a list using only the information given in a single list element. The procedure `Check_Inventory` calls `Check_One_Part` for each part in the list. Procedure `Check_One_Part` uses the information in a single part record to accomplish its task. What do we need to do to modify this code so that the iterator displays the names of all parts with less than 20 in stock? We need to change the literal 10 in procedure `Check_One_Part` to 20. Perhaps by using a variable in place of the literal, we could use this iterator to display the names of parts whose quantities in stock are less than some value supplied by the user. This approach is used in the following code fragment:

```
procedure Check_One_Part (Part : in out Part_Rec
 Cutoff : in Positive) is
begin
 if Part.Amount < Cutoff then
 Part_Strings.Put (Item => Part.Name);
 Text_IO.New_Line;
 end if;
end Check_One_Part;

-- The following instantiation does not compile
procedure Check_Inventory is new
 Auto_Part_List.Traverse (Process => Check_One_Part);
```

The Ada compiler does not accept the instantiation to create procedure `Check_Inventory`. The problem is that the parameter profile of `Check_One_Part` does not match that of the generic formal subprogram parameter `Process`. The number of parameters differ; `Process` has one parameter and `Check_One_Part` has two parameters. Should we change Specification 8-1 and add a second parameter to procedure `Process`? Such an approach is flawed because we cannot make the number and types of parameters for procedure `Process` general enough to handle *all* possible uses of the list. Instead we must find some way to allow `Check_One_Part` to access information outside of the element it is processing.

We solve this problem by writing `Check_One_Part` so that it accesses a global variable. You may have learned (correctly) that the use of global variables should be avoided to prevent side effects. However, like the global variables used in the implementation of ADOs (Chapter 3), we can restrict the scope of our global variables to minimize the potential for side effects. We do so by encapsulating these variables within a procedure as illustrated in the following code fragment:

```
procedure Check_Stock (Inventory : in Part_List;
 Cutoff : in Positive) is
-- This procedure displays the names of all parts in the
-- inventory with a quantity on hand less than Cutoff.

-- The "Process" procedure is nested within Check_Stock
procedure Check_One_Part (Part : in out Part_Rec) is
begin
 if Part.Amount < Cutoff then -- Cutoff is global to this procedure
 Part.Strings.Put (Item => Part.Name);
 Text_IO.New_Line;
 end if;
end Check_One_Part;

-- Instantiate the iterator
procedure Check_Inventory is new
 Auto_Part_List.Traverse (Process => Check_One_Part);
begin -- Check_Stock
 Check_All_Parts (List => Inventory); -- Call the iterator
end Check_Stock;
```

Figure 9-18 illustrates the declarative regions for procedure `Check_Stock`. This diagram shows that while `Cutoff` is global to procedure `Check_One_Part`, its scope still is restricted to the declarative region of procedure `Check_Stock`.

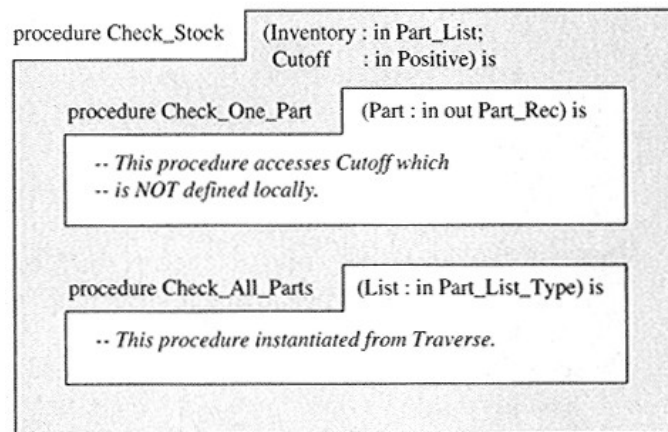


Figure 9-18  
Declarative Regions of Procedure `Check_Stock`

Another important use of this nesting technique is for applications that require us to use all the list elements to calculate a single value. For example, calculating sums and averages or finding maximum and minimum values. The following procedure calculates the average amount in stock of the parts in our inventory. Here, the processing of each element consists of incrementing a count and adding the amount in stock to a total:

```

procedure Find_Average (Inventory : in Part_List;
 Average : out Float) is
-- This procedure finds the average amount in stock for all parts.

 Count : Natural; -- These two variables are accessed
 Sum : Natural; -- globally by Process_One_Part.

 procedure Process_One_Part (Part : in out Part_Rec) is
 begin
 Count := Count + 1; -- Increment Count
 Sum := Sum + Part.Amount; -- Add amount in stock to Sum
 end Process_One_Part;

 -- Instantiate the iterator
 procedure Process_All_Parts is new
 Auto_Part_List.Traverse (Process => Process_One_Part);

begin -- Find_Average
 Count := 0; -- Initialize
 Sum := 0;
 Process_All_Parts (List => Inventory); -- Call the iterator
 Average := Float (Sum) / Float (Count); -- Calculate the average
end Find_Average;

```

As in the previous example, instantiation allows us to easily create operations that access every element in a list. The procedure nesting allows us to control the visibility of "global" variables that our processing operation must access to accomplish our particular task. We use this technique to instantiate similar traversal procedures in later chapters.

Ada 95 provides pointers to subprograms as an alternative technique for supplying a traversal procedure with a process procedure. We can pass a pointer to a different process procedure with each different call of a nongeneric traverse operation. Pointers to subprograms are not available in Ada 83; we must use the techniques described in this section.

### Lists with Duplicate Elements

The lists we have discussed so far in this chapter and Chapter 8 were all composed of elements with unique keys; that is, there were no duplicate keys in the list. For instance, a list of university students might be made up of elements of the following type:

```

type Student_Rec is
 record
 ID : Positive;
 Last_Name : Name_Str;
 First_Name : Name_Str;
 GPA : Float;
 end record;

```

The list might be ordered according to Student ID. In this case, there would be no duplicate keys in the list, as each student has a unique ID number [Figure 9-19(a)]. However, if the list was instantiated for assigning class rank, and GPA was the key on which the elements were ordered, there would almost certainly be some duplicate keys

[Figure 9-19(b)]. Duplicate keys are also possible if the list is to be ordered alphabetically by the students' names [Figure 9-19(c)].

When lists are not ordered according to a unique key, the operations specified for the list must be changed to reflect this situation. Some of the changes would be internal to the operation. For example, should the Insert operation insert a new element before or after elements already in the list that have the same key as the new one?

Other changes, however, must be reflected in the specifications of the operations themselves. For instance, what is returned by Retrieve, which searches a list for an element with a specific key, when the keys are not unique? Does it return the first element with the specified key, or a list of the elements that have this key? What is done by Delete, which takes a list and the key of the element to delete, if there are multiple elements with that key? Should they all be deleted? Should the procedure specification be changed to take a list and copy of the element (rather than just its key value) to make sure that the procedure deletes only a particular element? Similar questions can be raised for the Modify operation. These are no longer purely internal implementation issues, since the application programmer must understand the interfaces and the functions of the operations.

We do not tell you a single algorithmic answer to the problem of elements with duplicate keys, for the solution depends on how the list will be used. These are design choices that you, as the programmer, must make and record in the specification of the list class. In the programs that you are writing now, you are probably wearing two hats: one as the application programmer using the list class and one as the list implementer. In many real-world programming situations, you will use classes that another programmer has implemented. Good communications skills (and well-documented specifications) are essential for the programmers who write "utilities" that other programmers use.

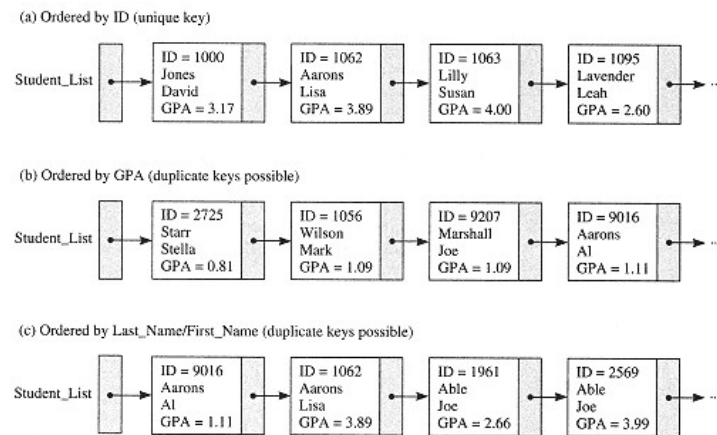


Figure 9-19  
Duplicate Keys in a List

### Case Study: Unbounded Natural Numbers

The range of integer values is limited. For instance, if an integer variable is stored in 16 bits, integer values can only range from -32,768 to +32,767. If 32 bits are used, the range is still only from -2,147,483,648 to 2,147,483,647. This does not mean that we cannot do arithmetic with whole numbers outside of these ranges. We can implement our own integer type with an unbounded range. Specification 9-4 describes an unbounded number class with operations for addition, subtraction, multiplication, and division of non-negative integers. It also includes procedures for assigning (Copying) and for the input and output of these unbounded numbers. There are many other operations we could have defined for this class including the relational operators and operations to convert between these unbounded numbers and Ada's predefined types. We leave these as exercises. Here is a simple application program that uses our unbounded number class:

```
with Unbounded_Natural;
with Text_IO;
procedure Test_Add is
-- A simple program that uses the Unbounded Natural class
 A : Unbounded_Natural.Natural;
 B : Unbounded_Natural.Natural;
 C : Unbounded_Natural.Natural;
begin
 Text_IO.Put_Line ("Enter two big numbers");
 Unbounded_Natural.Get (Item => A);
 Unbounded_Natural.Get (Item => B);
 Text_IO.Put_Line ("The sum of");
 Unbounded_Natural.Put (Item => A, Width => 25);
 Text_IO.New_Line;
 Unbounded_Natural.Put (Item => B, Width => 25);
 Text_IO.New_Line;
 Text_IO.Put_Line ("is");
 Unbounded_Natural.Add (Left => A, Right => B, Result => C);
 Unbounded_Natural.Put (Item => C, Width => 25);
 Text_IO.New_Line;
end Test_Add;
```

### Specification 9-4— Unbounded Natural Numbers

```
with Text_IO;
package Unbounded_Natural is

-- This package implements a class of unbounded non-negative integers.
-- The size of each integer is limited only by the amount of memory.

 type Natural is limited private;

 DATA_ERROR : exception;
```

```

procedure Assign (Target : in out Natural;
 Source : in Natural);
-- Make Target a copy of Source, i.e., Target := Source

procedure Add (Left : in Natural;
 Right : in Natural;
 Result : in out Natural);
-- Result := Left + Right

procedure Subtract (Left : in Natural;
 Right : in Natural;
 Result : in out Natural);
-- Result := Left - Right

-- Exception : CONSTRAINT_ERROR is raised if Right > Left.

procedure Multiply (Left : in Natural;
 Right : in Natural;
 Result : in out Natural);
-- Result := Left * Right

procedure Divide (Left : in Natural;
 Right : in Natural;
 Result : in out Natural);
-- Result := Left / Right

-- Exception : CONSTRAINT_ERROR is raised if Right is zero.

procedure Get (File : in Text_IO.File_Type := Text_IO.Current_Input;
 Item : in out Natural);
-- Gets a natural number from the File. Skips over leading blanks and line
-- terminators. Gets digits until end of line or a nondigit is
-- encountered. Unlike Integer_IO.Get, this procedure consumes the
-- nondigit that marks the end of the number.

-- Exceptions : DATA_ERROR is raised if the first nonblank character
-- found is not a digit.

procedure Put (File : in Text_IO.File_Type := Text_IO.Current_Output;
 Item : in Natural;
 width : in Standard.Natural);
-- Puts the natural number to a file.

```

```

procedure Finalize (Object : in out Natural);
-- Release all the memory associated with an unbounded natural number.

-- Preconditions : None
--
-- Postconditions : The memory used by Object is released. Object is undefined.

private

type Node_Type;

type Natural is access Node_Type;
subtype Node_Ptr is Natural; -- Synonym for the pointer type

subtype Digit_Type is Integer range 0..9;

type Node_Type is
 record
 Digit : Digit_Type;
 Back : Node_Ptr;
 Next : Node_Ptr;
 end record;

end Unbounded_Natural;

```

### ***The Design***

What sort of data structure could we use to implement this class? We had a similar problem with unbounded-length strings in Chapter 5. There we implemented a string as a pointer to an unconstrained array of characters. We could use a similar data structure to solve this problem; using a pointer to an unconstrained array of digits instead of characters. An alternative data structure is a linked list of digits. Because this chapter is about linked lists, that is the data structure we use here.

In the simplest case, each node in the list represents one digit in the number. Figure 9-20(a) shows one possible ordering of digits in a number—the most significant digit is first in the list and the least significant digit is last. Displaying a number stored as such a list is easy; we traverse the list from beginning to end displaying the digit when we process the node. However, storing the digits in this order makes the arithmetic operations complex. Figure 9-20(b) shows the addition of 83,536 and 41. We need to start adding with the digits at the ends of each list (6 + 1). To simplify arithmetic operations, it would be more convenient to store the digits in reverse order, as shown in Figure 9-20(c). But this ordering makes the display operations more complex.

We need to process the digits in one direction (most to least significant) to display a number and in the other direction to add them. Because we can process the digits in either order, the doubly linked list is an ideal structure for representing our long integers. Using a circular list allows us to easily begin processing at either end of the list. Figure 9-20(d) shows how the number 567,890 is stored in a circular doubly linked list.

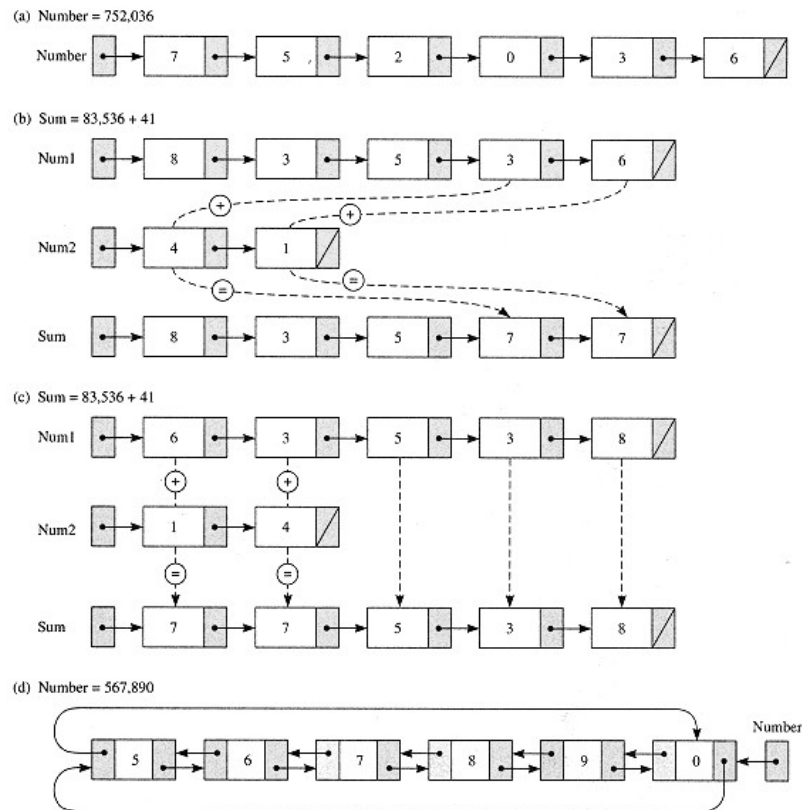


Figure 9-20  
Representing Large Integers with Linked Lists

### The Implementation

The types to implement our unbounded number as a circular doubly linked list are shown in the private parts of Specification 9-4. Except for the name for the information field, these types are taken directly from the circular doubly list implementation of the sorted list we discussed earlier in this chapter. Body 9-1 contains the implementation of all the operations defined in Specification 9-4.

To display a number, we use the traversal algorithm for circular doubly linked lists we discussed earlier. Our processing of each node consists of displaying the digit stored there. The Get operation requires a little more work. We read the digits one at a time as characters with `Text_IO.Get` and convert each to a numeric digit. We insert each digit at the end of our list. Here is our algorithm:



**Get**

```

skip over any leading blanks and line terminators
loop
 exit when no more digits remain
 get a character
 convert the character to a digit
 insert digit at end of list
end loop

```

The exit condition of the loop in this algorithm is an abstraction of two conditions: no more digits remaining when we reach the end of the line (found a line terminator) or when we find a character other than one of the ten digits. We must test for the first condition before getting a character and for the second condition after getting a character.

To add two unbounded numbers, we use the algorithm you learned when you first added multidigit numbers. Here's an example of adding 8,726 and 1,557:

|           |   |         |
|-----------|---|---------|
| 1 1 0 1   | ← | Carries |
| 8 7 2 6   |   |         |
| + 1 5 5 7 |   |         |
| 1 0 2 8 3 |   |         |

First we add the digits in the one's column (the rightmost column of the number). In our example, the result is 13. The 3 becomes the rightmost digit in the answer and the 1 is carried over to the ten's column. Now we add the 10's column (1 + 2 + 5) giving an answer of 8 and no carry. We continue until there are no more columns remaining in either number. In our Ada implementation, we use a pointer to traverse the digits of each number from right to left. The sum of every pair of digits (and the possible carry from the previous column) is inserted at the beginning of the answer list.

The algorithm we use for subtraction also begins with the digits in the rightmost column. If the digit in the second number is greater than that in the first, we must make a borrow from the next most significant column. Our subtraction procedure raises `CONSTRAINT_ERROR` if the result would be negative. We know that the result is negative if, after traversing all the digits in the first number, there are still digits remaining in the second or there is still a borrow remaining.

Our multiplication algorithm is based on the method used to multiply multidigit numbers by hand:

|             |
|-------------|
| 7 3 2       |
| × 1 6 5     |
| 3 6 6 0     |
| 4 3 9 2     |
| 7 3 2       |
| 1 2 0 7 8 0 |

We multiply the first number by each digit of the second. Each of these partial products is shifted to the left and then summed. In our procedure, shifting a number to the left is accomplished by inserting the appropriate number of zero digits at the end of its list of digits.

Our algorithm for division is very simple. We count the number of times that we can subtract the divisor from the dividend. We exit the loop when the divisor is greater than the *remaining* dividend. Because we have left the implementation of the greater than operator as an exercise, we used the exception generated by the subtraction operation to terminate

our loop. In general, we prefer to reserve exceptions for exceptional events. We expect the divisor in the loop to become greater than the remaining dividend; this condition is not an exceptional event.

Our division method is not very efficient. For example, dividing 1,000,000 by 2 requires half a million subtraction operations. The algorithm you learned for doing long division is much more efficient. We leave its implementation for unbounded numbers as a programming problem.

Body 9-1 also includes five local procedures called by the procedures that implement the specified operations. Procedure `Append` inserts a digit at the end of a linked list of digits and procedure `Prefix` inserts a digit at the beginning of a linked list of digits. Procedure `Remove_Zeros` deletes any leading zeros in the linked list of digits. Function `Length` returns the number of nodes in the linked list. And finally, the local procedure `Multiply` determines the product of an unbounded natural number and a single digit.

### Body 9-1—

#### Unbounded Natural Numbers Implements Specification 9-4

```
with Integer_IO;
with Unchecked_Deallocation;
package body Unbounded_Natural is

-- This package implements a class of unbounded non-negative integers.
-- The size of each integer is limited only by the amount of memory.

-- Operations local to package body

-- Instantiate a procedure to recycle memory
procedure Free is new Unchecked_Deallocation (Object => Node_Type,
 Name => Node_Ptr);

function Length (List : in Node_Ptr) return Standard.Natural is
-- Returns the number of nodes in the list
 Count : Standard.Natural; -- Node counter
 Current : Node_Ptr; -- Traversal pointer
begin
 Count := 0;
 if List /= null then
 Current := List;
 loop -- Each iteration, one node is counted
 Count := Count + 1;
 Current := Current.all.Next;
 exit when Current = List;
 end loop;
 end if;
 return Count;
end Length;
```

```

procedure Remove_Zeros (List : in out Node_Ptr) is
-- Removes any leading zeros in the list making up a number
 First_Digit : Node_Ptr; -- Pointer to node with most significant digit
begin
 loop -- Each iteration, one leading zero is removed
 First_Digit := List.all.Next; -- Pointer to most significant digit
 exit when First_Digit.all.Digit /= 0 or -- Leading digit is not zero
 First_Digit = List; -- Don't remove last digit
 -- Delete the most significant digit (it is a zero)
 List.all.Next := First_Digit.all.Next; -- Change link in preceding node
 First_Digit.all.Next.Back := List; -- Change link in next node
 Free (First_Digit); -- Recycle memory
 end loop;
end Remove_Zeros;

```

```

procedure Append (List : in out Node_Ptr;
 Digit : in Digit_Type) is
-- Appends a digit to the least significant end of the list of digits

```

```

-- Preconditions : None
--
-- Postconditions : List = List & Digit

```

```

begin
 if List = null then
 -- Add new node to the empty list
 List := new Node_Type;
 List.all := (Digit => Digit, Back => List, Next => List);
 else -- Add new node to the end of the list
 List.all.Next := new Node_Type'(Digit => Digit,
 Back => List,
 Next => List.all.Next);
 -- Adjust list pointer to designate last node in list (the new one)
 List := List.all.Next;
 -- Adjust the first node's back pointer
 List.all.Next.all.Back := List;
 end if;
end Append;

```

```

procedure Prefix (List : in out Node_Ptr;
 Digit : in Digit_Type) is
-- Prefixes a digit to the most significant end of the list of digits

```

```

-- Preconditions : None
--
-- Postconditions : List = Digit & List

```

```

New_Ptr : Node_Ptr; -- Designates a new node
begin

```

```

if List = null then
 -- Add new node to the empty list
 List := new Node_Type;
 List.all := (Digit => Digit, Back => List, Next => List);
else -- Add new node to the beginning of the list
 List.all.Next := new Node_Type'(Digit => Digit,
 Back => List,
 Next => List.all.Next);
 -- Change the Back pointer of the original "first" node
 List.all.Next.all.Back := List.all.Next;
end if;
end Prefix;

procedure Multiply (Value : in Natural;
 Digit : in Digit_Type;
 Result : in out Natural) is
-- Multiplies an unbounded natural number by a single digit

 subtype Carry_Range is Integer range 0..9; -- Range of carry
 subtype Product_Range is Integer range 0..90; -- Range of product

 Product : Product_Range; -- Product of two digits plus carry
 Carry : Carry_Range; -- Carry resulting from product of two digits
 Answer : Natural; -- Local copy of Result

 Value_Ptr : Node_Ptr; -- Pointer to traverse Value

begin
 Value_Ptr := Value; -- Start with rightmost digit of value
 Carry := 0; -- No carry to add to first product
 Product_Loop:
 loop -- Each iteration, one digit of Value is multiplied by Digit
 Product := Carry + Digit * Value_Ptr.all.Digit;
 -- Handle any carry to next digit
 Carry := Product / 10;
 Product := Product rem 10;
 -- Insert product at the beginning of the linked list
 Prefix (List => Answer, Digit => Product);
 -- Advance to next most significant digit
 Value_Ptr := Value_Ptr.all.Back;
 exit Product_Loop when Value_Ptr = Value;
 end loop Product_Loop;
 -- Handle any remaining carry
 if Carry /= 0 then
 Prefix (List => Answer, Digit => Carry);
 end if;
 Remove_Zeros (Answer);

 Finalize (Result); -- Recycle any memory used by old unbounded number
 Result := Answer; -- Assign local Answer to Result
end Multiply;

```

```

-- Operations defined in package specification

procedure Assign (Target : in out Natural;
 Source : in Natural) is
-- This procedure should handle the degenerate case where Source = Target.
 Current : Node_Ptr; -- Traversal Pointer
 Result : Natural; -- Local copy of Target
begin
 Current := Source; -- Start with least significant digit
 loop
 Prefix (List => Result, Digit => Current.all.Digit); -- Copy digit
 Current := Current.all.Back; -- Move to next most significant digit
 exit when Current = Source; -- Done when we get back to start
 end loop;
 Finalize (Target); -- Recycle any memory used by old Target
 Target := Result; -- Assign local Result to Target
end Assign;

```

```

procedure Add (Left : in Natural;
 Right : in Natural;
 Result : in out Natural) is

 subtype Carry_Range is Integer range 0..1; -- Range of carry
 subtype Sum_Range is Integer range 0..19; -- Range of sum

 Left_Ptr : Node_Ptr; -- For traversing the
 Right_Ptr : Node_Ptr; -- two lists
 Sum : Sum_Range; -- Result of adding
 Carry : Carry_Range; -- a single digit
 Answer : Natural; -- Local copy of Result

begin
 -- Initialize pointers to traverse the two lists in reverse order
 -- (from least to most significant digit)
 Left_Ptr := Left;
 Right_Ptr := Right;
 Carry := 0; -- No carry for the ones column

 Sum_Loop:
 loop -- Each iteration, one digit of the Answer is calculated
 Sum := Carry + Left_Ptr.all.Digit + Right_Ptr.all.Digit;
 -- Handle any carry to the next digit
 Carry := Sum / 10;
 Sum := Sum rem 10;
 -- Insert the sum at the beginning of the linked list
 Prefix (List => Answer, Digit => Sum);
 -- Advance pointers to next digit
 Left_Ptr := Left_Ptr.all.Back;
 Right_Ptr := Right_Ptr.all.Back;
 -- Stop when we run out of digits in one of the terms
 exit Sum_Loop when Left_Ptr = Left or Right_Ptr = Right;
 end loop Sum_Loop;

```

```

-- If one number is longer than the other number, there are some
-- extra leading digits that haven't been added to the Answer.

Transfer_Left: -- Transfer any remaining digits from Left
loop -- Each iteration, one remaining digit in Left is transferred to Answer
 exit Transfer_Left when Left_Ptr = Left;
 Sum := Carry + Left_Ptr.all.Digit;
 -- Handle any carry to the next digit
 Carry := Sum / 10;
 Sum := Sum rem 10;
 Prefix (List => Answer, Digit => Sum);
 -- Advance pointer to next digit
 Left_Ptr := Left_Ptr.all.Back;
end loop Transfer_Left;

Transfer_Right: -- Transfer any remaining digits from Right
loop -- Each iteration, one remaining digit in Right is transferred to Answer
 exit Transfer_Right when Right_Ptr = Right;
 Sum := Carry + Right_Ptr.all.Digit;
 -- Handle any carry to the next digit
 Carry := Sum / 10;
 Sum := Sum rem 10;
 Prefix (List => Answer, Digit => Sum);
 -- Advance pointer to next digit
 Right_Ptr := Right_Ptr.all.Back;
end loop Transfer_Right;

-- There still might be a carry remaining
if Carry /= 0 then
 Prefix (List => Answer, Digit => Carry);
end if;

Finalize (Result); -- Recycle any memory used by old unbounded number
Result := Answer; -- Assign local Answer to Result
end Add;

procedure Subtract (Left : in Natural;
 Right : in Natural;
 Result : in out Natural) is

 subtype Borrow_Range is Integer range 0..1; -- Range of borrow
 subtype Difference_Range is Integer range 0..9; -- Range of difference

 Left_Ptr : Node_Ptr; -- For traversing the
 Right_Ptr : Node_Ptr; -- two lists

 Difference : Difference_Range; -- Result of subtracting
 Borrow : Borrow_Range; -- a single digit
 Answer : Natural; -- Local copy of Result

```

```

begin
 -- Initialize pointers to traverse the two lists in reverse order
 -- (from least to most significant digit)
 Left_Ptr := Left;
 Right_Ptr := Right;
 Borrow := 0; -- No borrow from the ones column

 Diff_Loop:
 loop -- Each iteration, one digit of the result is calculated
 if Left_Ptr.all.Digit - Right_Ptr.all.Digit - Borrow < 0 then
 -- Need to borrow from next column to subtract the digits
 Difference := 10 + Left_Ptr.all.Digit - Borrow - Right_Ptr.all.Digit;
 Borrow := 1; -- To be subtracted from next column
 else
 -- No need to borrow to subtract the digits
 Difference := Left_Ptr.all.Digit - Borrow - Right_Ptr.all.Digit;
 Borrow := 0; -- No borrow to be subtracted from next column
 end if;
 -- Insert the difference at the beginning of the linked list
 Prefix (List => Answer, Digit => Difference);
 -- Advance pointers to next digit
 Left_Ptr := Left_Ptr.all.Back;
 Right_Ptr := Right_Ptr.all.Back;
 -- Stop when we run out of digits in one of the terms
 exit Diff_Loop when Left_Ptr = Left or Right_Ptr = Right;
 end loop Diff_Loop;

 -- If there are still digits remaining in Right, Answer would be negative.
 if Right_Ptr /= Right then
 raise CONSTRAINT_ERROR; -- Negative Result is out of range of Natural
 end if;
 -- Take care of any remaining digits in Left
 Transfer_Left:
 loop -- Each iteration, one remaining digit in Left is transferred to Answer
 exit Transfer_Left when Left_Ptr = Left;
 if Left_Ptr.all.Digit < Borrow then
 -- Need to borrow from next column to subtract the digits
 Difference := 10 + Left_Ptr.all.Digit - Borrow;
 Borrow := 1; -- To be subtracted from next column
 else
 -- No need to borrow to subtract the digits
 Difference := Left_Ptr.all.Digit - Borrow;
 Borrow := 0; -- No borrow to be subtracted from next column
 end if;
 Prefix (List => Answer, Digit => Difference);
 -- Advance pointer to next digit
 Left_Ptr := Left_Ptr.all.Back;
 end loop Transfer_Left;

 -- If there is still an outstanding borrow, the result is negative.
 if Borrow /= 0 then
 raise CONSTRAINT_ERROR; -- Negative Result is out of range of Natural
 end if;

```

```

 Remove_Zeros (Answer); -- Remove any leading zeros from Answer
 Finalize (Result); -- Recycle any memory used by old unbounded number
 Result := Answer; -- Assign local Answer to Result
end Subtract;

```

```

procedure Multiply (Left : in Natural; -- Multiplicand
 Right : in Natural; -- Multiplier
 Result : in out Natural) is
-- Uses "grade school" multiplication algorithm

```

```

 Right_Ptr : Node_Ptr; -- For traversing Right
 Shift : Standard.Natural; -- How far to shift a partial sum

```

```

 Term : Natural; -- The multiplicand * one digit in the multiplier
 Sum : Natural; -- Sum of the terms (the answer)
begin
 Append (List => Sum, Digit => 0); -- Initialize Sum to zero
 Shift := 0;
 Right_Ptr := Right; -- Begin with the least significant digit
 Multiplier_Loop:
 loop -- Each iteration, one digit of the multiplier is processed
 Multiply (Value => Left,
 Digit => Right_Ptr.all.Digit,
 Result => Term);
 -- Shift the term to the right
 for I in 1..Shift loop
 Append (List => Term, Digit => 0);
 end loop;
 -- Add the term to the Sum
 Add (Left => Sum,
 Right => Term,
 Result => Sum);
 -- Advance to the next digit in the multiplier
 Right_Ptr := Right_Ptr.all.Back;
 exit Multiplier_Loop when Right_Ptr = Right;
 Shift := Shift + 1; -- Increment shift for next term
 end loop Multiplier_Loop;

```

```

 Result := Sum; -- Assign local answer to Result
end Multiply;

```

```

procedure Divide (Left : in Natural;
 Right : in Natural;
 Result : in out Natural) is
-- The algorithm for Divide is to count how many times Right can be
-- subtracted from Left. We can't destroy Left, so a copy (Remaining) is
-- made first. The count is kept as an unbounded natural and incremented
-- by calling Add to add one to it.
 Count : Natural; -- Number of times Right is subtracted from left
 One : Natural; -- The value 1 (as an unbounded Natural)
 Remaining : Natural; -- Result of subtracting Right

```



```

begin
 -- Are we trying to divide by zero?
 if Right = Right.all.Next and Right.all.Digit = 0 then
 raise CONSTRAINT_ERROR;
 end if;

 Append (List => One, Digit => 1); -- Initialize "constant"
 Append (List => Count, Digit => 0); -- Initialize Count to zero
 Assign (Target => Remaining, Source => Left); -- Make a copy of Left

 Subtract_Block: -- Block for exception handler
 begin
 loop
 -- Exit when Subtract raises CONSTRAINT_ERROR. This happens
 -- when Right is greater than Remaining
 Subtract (Left => Remaining, -- Subtract Right
 Right => Right,
 Result => Remaining);
 -- Increment subtraction count
 Add (Left => Count, Right => One, Result => Count);
 end loop;
 exception
 when CONSTRAINT_ERROR =>
 null; -- Nothing to do, Count contains the answer
 end Subtract_Block;

 -- Recycle memory used by local variables
 Finalize (One);
 Finalize (Remaining);

 Finalize (Result); -- Recycle any memory used by old unbounded number
 Result := Count; -- Count contains the result
end Divide;

procedure Get (File : in Text_IO.File_Type := Text_IO.Current_Input;
 Item : in out Natural) is
 subtype Digit_Char is Character range '0'..'9';
 Char : Character; -- A character from the input file
 Digit : Digit_Type; -- Numeric equivalent of Char
begin
 Finalize (Item); -- Recycle any memory used by old unbounded number
 -- Skip over any leading blanks and leading line terminators
 Skip_Loop:
 loop -- Each iteration, one leading blank is processed
 Text_IO.Get (File => File, Item => Char);
 exit Skip_Loop when Char /= ' ';
 end loop Skip_Loop;
 -- Is the first character a digit? It should be.
 if Char not in Digit_Char then
 raise DATA_ERROR;
 end if;
 -- Assertion: Char holds the first digit of the number

```

```

Digit_Loop:
loop
 -- Convert the character to a numeric digit
 Digit := Character'Pos(Char) - Character'Pos('0');
 -- Add it to the end of the list
 Append (List => Item, Digit => Digit);
 exit Digit_loop when Text_IO.End_Of_Line (File); -- Line terminator?
 Text_IO.Get (File => File, Item => Char);
 exit when Char not in Digit_Char; -- Non digit?
end loop Digit_Loop;
-- Remove any leading zeros from the list
Remove_Zeros (Item);
end Get;

procedure Put (File : in Text_IO.File_Type := Text_IO.Current_Output;
 Item : in Natural;
 Width : in Standard.Natural) is
 Current : Node_Ptr; -- For traversing the list
begin
 -- Display necessary leading blanks for the given Width
 for I in 1..Width-Length(Item) loop
 Text_IO.Put(' ');
 end loop;
 -- Put all the digits
 Current := Item.all.Next; -- Set Current to 1st item in list
 loop -- Each iteration, one digit is written
 -- Put the current digit
 Integer_IO.Put (File => File, Item => Current.all.Digit, Width => 1);
 exit when Current = Item;
 Current := Current.all.Next;
 end loop;
end Put;

procedure Finalize (Object : in out Natural) is
 To_Recycle : Node_Ptr;
 First : Node_Ptr; -- Pointer to 1st node in the list
begin
 if Object /= null then
 First := Object.all.Next; -- Set First to 1st node in list
 Object.all.Next := null; -- Break the circular link (to end loop)
 loop -- Each iteration, one node is recycled
 exit when First = null;
 To_Recycle := First;
 First := First.all.Next;
 Free (To_Recycle);
 end loop;
 Object := null;
 end if;
end Finalize;

end Unbounded_Natural;

```

## Summary

In this chapter we have looked at a number of variations on the linked list theme. Obviously, given the wide variety of applications that use lists, there are many interesting ways that a programmer can use this way of organizing data. All of the implementations are supported by a common method of development: As programmers, we determine the needs of the application and then decide on a physical representation for the data. Along with the declarations of the data type, we must provide a set of basic operations so that the user of the data type can create and access elements in the structure. For lists, these operations generally include procedures and functions to add, delete, and modify elements, traverse a list, and determine whether the list is empty. We don't expect—or allow—the application programmer to manipulate the data structure directly.

The idea of linking the elements in a list has been extended in this chapter to include lists with header and trailer nodes, circular lists, and doubly linked lists. The idea of linking the elements is a possibility to consider in the design of many types of data structures.

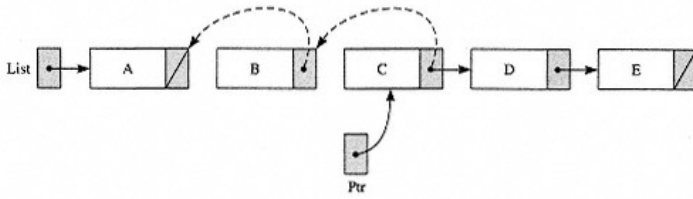
The dynamic allocation of the space for nodes in a linked structure allows the program to get just what it needs during execution. In a program where the amount of data to be stored is very unpredictable or may vary widely, the dynamic allocation of space has major advantages. An access variable provides efficient access to a node, for it contains (among other information) the node's actual address in memory.

While a linked list can be used to implement virtually any list application, its real strength is in applications that largely process the list elements in order. This is not to say that we cannot do "random access" operations on a linked list. Our List class specifications include operations that access elements in random order—for instance, procedures `Retrieve`, `Modify`, and `Delete` manipulate a particular element in the list. However, the only way to find an element is to search the list, beginning at the first element, and continuing sequentially to examine element after element. This search is  $O(N)$ , since the amount of work required is directly proportional to the number of elements in the list. A particular element in a sequentially ordered list in an array, in contrast, can be found with a binary search, decreasing the search algorithm to  $O(\log_2 N)$ . For a large list, the  $O(N)$  sequential search can be quite time-consuming. There is a linked structure that supports  $O(\log_2 N)$  searches: the binary search tree. We discuss this data structure in detail in Chapter 12.

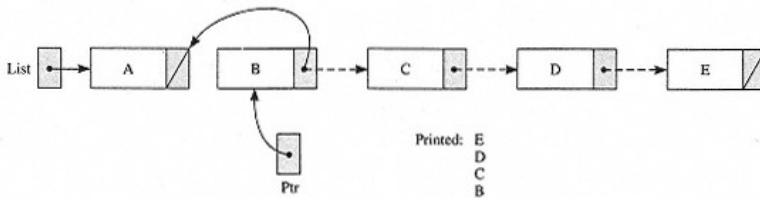
## Exercises

1. The element in a trailer node must have a key that is greater than any legitimate key in the list. Why doesn't the element in the header node have to have a key that is less than any legitimate key in the list?

2. (a) Write a procedure, `Print_Reverse`, that takes the external pointer to a singly linked list and prints the elements in reverse order using the following approach: Traverse the list from first to last node, reversing the direction of each `Next` pointer, as illustrated below:



When you reach the end of the list, you have a linked list whose elements are in the reverse order from the original list. Now traverse this list, processing each element by printing the element and re-inverting its `Next` pointer, as illustrated below:



When you reach the (original) beginning of the list, the whole list is printed in reverse order, and the links are back to their original values.

- (b) What is the Big-O of this algorithm? How many times is each node accessed?
3. Dummy nodes are used to simplify list processing by eliminating some "special case."
- What special case is eliminated by a header node in a linear linked list?
  - What special case is eliminated by a trailer node in a linear linked list?
  - Would dummy nodes be useful in implementing a linked stack? That is, would their use eliminate a special case?
  - Would dummy nodes be useful in implementing a linked queue with a pointers to both head and rear elements?
  - Would dummy nodes be useful in implementing a circular linked queue?
4. Complete the procedure bodies for the remaining operations for the linked list with header and trailer nodes in this chapter: `Clear`, `Full`, `Retrieve`, and `Modify`.
5. Complete the procedure bodies for the remaining operations for the circular linked list in this chapter: `Clear`, `Empty`, `Full`, `Retrieve`, and `Modify`.
6. Implement the FIFO queue ADT as a circular linked list, with the external pointer accessing the "rear" node of the queue.
7. Write a procedure `Print_Reverse` that takes a pointer to the *first* element in a circular singly linked list and prints out the elements in reverse order. For instance, for the list `X Y Z`, `Print_Reverse(List)` would output `Z Y X`. (Hint: Use a stack of access values. You may instantiate a stack package from the stack specification used in Chapter 6.)

8. The following procedure declaration is given in the package declaration of a sorted list ADT:

```

procedure Merge (List1 : in out List_Type;
 List2 : in out List_Type;
 New_List : out List_Type);

-- Merge List1 and List2 to produce New_List.
--
-- Preconditions : None
--
-- Postconditions : New_List contains the elements of List1 and
-- List2, with no duplicate keys. If a key
-- exists in both List1 and List2, the element
-- from List1 is put into New_List.
--
-- List1 and List2 are empty.

```

(a) Write the body for this procedure for the sorted list implementation given in this chapter that uses a linear linked list with header and trailer nodes.

(b) Write the body for this procedure for the sorted list implementation given in this chapter that uses a circular linked list.

9. A list is implemented as a circular singly linked list, with a header node containing an impossibly small value as its key (supplied through the generic formal object `Min_Element`). The node accessed by the external pointer `List` is considered the "current" node in the list.

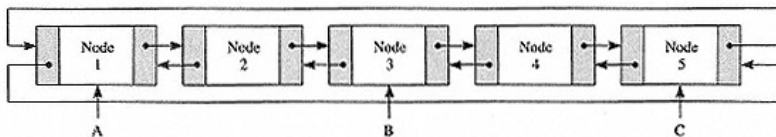
(a) Write the body for procedure `Reset`, which takes the `List` pointer and advances it to access the first node in the list (the one following the header node).

(b) Write the body for procedure `Retrieve_Current`, which takes the `List` pointer and returns both the node's `Info` part and a Boolean flag indicating whether the end of the list was reached (that is, if the current node is the header). This procedure should also advance the `List` pointer to access the node that succeeds the current node, leaving the list ready for another call to `Retrieve_Current` to retrieve the next element.

(c) Write the body for generic procedure `Traverse`, which uses `Reset` and `Retrieve_Current` to access the list data. Call procedure `Process` for every element in the list.

10. The Big-O of the delete procedure we developed for a circular linked list is  $O(1)$  for deleting the element at the beginning of the list and  $O(N)$  for deleting the element at the end of the list. Can we change the search and/or delete procedures so that deletion of first and last elements is  $O(1)$ ?

11. Using the circular doubly linked list below, give the expression corresponding to each of the following descriptions.



(For example, the expression for the `Info` field of Node 1, referenced from pointer A, would be `A.all.Info`.)

(a) The `Info` field of Node 1, referenced from pointer C

(b) The `Info` field of Node 2, referenced from pointer B

(c) The `Next` field of Node 2, referenced from pointer A

(d) The `Next` field of Node 4, referenced from pointer C

- (e) Node 1, referenced from pointer B
- (f) The Back field of Node 4, referenced from pointer C
- (g) The Back field of Node 1, referenced from pointer A

12. Modify the `Create` procedure for the doubly linked list with header and trailer nodes on page 403 so that there is no precondition. That is, allow the application programmer to use `Create` to set a list to empty whether it has been previously initialized or not.

13. Figure 9-17 illustrates a doubly linked list with a single dummy node. Figure 9-17(c) shows an empty list for this implementation.

- (a) Does this implementation require a `Create` procedure to initialize a list to empty or can we supply default initial values in the `List_Type` record to accomplish the desired initialization?
- (b) Write the code (the `List_Type` record or the specification and body of procedure `Create`) necessary to initialize a list.

14. The text edited by a line editor is represented by a doubly linked list of nodes, each of which contain an 80-column line of text (`Line_Type`). There is one external pointer (`Line_Ptr_Type`) to this list, which points to the "current" line position in the text being edited. The list has a header node, which contains the string

```
---Top of File---
```

and a trailer node, which contains the string

```
---Bottom of File---
```

- (a) Draw a sketch of this data structure.
- (b) Write the type declarations to support this data structure.
- (c) Code the following operations on this structure.

```
procedure Create (Line_Ptr : in out Line_Ptr_Type);
```

```
procedure Go_To_Top (Line_Ptr : in out Line_Ptr_Type);
-- Line_Ptr is set to access "first" line.
```

```
procedure Go_To_Bottom (Line_Ptr : in out Line_Ptr_Type);
-- Line_Ptr is set to access "last" line.
```

- (d) Describe `Go_To_Top` and `Go_To_Bottom` in terms of Big-O. How could the list be changed to make these operations  $O(1)$ ?
- (e) Code the `Insert_Line` operation, using the following declaration:

```
procedure Insert_Line (Line_Ptr : in Line_Ptr_Type;
 New_Line : in Line_Type);
-- Insert New_Line after the "current" line.
```

- (f) Code the `Display_Text` operation, using the following declaration:

```
procedure Display_Text (Line_Ptr : in out Line_Ptr_Type);
-- Displays all the lines from "first" to "last".
```

- (g) After the call, `Display_Text(Text_Line)`, which node does `Line_Ptr` access?

15. Of the three variations of linked lists (circular, with header and trailer nodes, and doubly linked) which would be most appropriate for each of the following applications?

- (a) Search a list for a key and return the keys of the two elements that come before it and the keys of the two elements that come after it.
- (b) A text file contains integer elements, one per line, *ordered* from smallest to largest. You must read the values from the file and create an ordered linked list containing the values.
- (c) A short list frequently becomes empty. You want a list that is optimum for inserting an element into the empty list and deleting the only element from the list.

16. Using the declarations for the auto part list given on page 406, write a procedure that returns the name of the part with the greatest amount in stock. Make use of the generic list traversal procedure in your procedure.

17. Complete the following local search procedure for a sorted list *that can contain duplicate keys*. The list is implemented as a singly linked list with header and trailer nodes.

```

procedure Search_Linked_List (List : in List_Type;
 Key : in Key_Type;
 Found : out Boolean;
 Pred_Loc : out Node_Ptr;
 Location : out Node_Ptr) is
-- Searches for the location of Key in the List

-- Preconditions : The nodes in List are in ascending order.
-- List has Header and Trailer Nodes.

-- Postconditions : If Key is in List
-- Found is True.
-- Location designates the first element with
-- Key.
-- Pred_Loc designates the predecessor of the
-- node designated by Location.
-- else
-- Found is False.
-- Location designates the node that would
-- follow a node containing Key.
-- Pred_Loc designates the predecessor of the
-- node designated by Location.

```

18. The following is a declaration for an Insert operation for a sorted list *that can contain duplicate keys*.

```

procedure Insert (List : in out List_Type;
 Item : in Element_Type);
-- Insert Item into List
--
-- Preconditions : None
--
-- Postconditions : Item is inserted into List. If one or more
-- elements with the key of Item are already in
-- the list, Item is inserted after the last of
-- these "duplicate" elements.

```

The list is implemented as a singly linked list with header and trailer nodes. Write the procedure body for this operation. You may call the search procedure whose specification was given in Exercise 17.

19. The following is a declaration for a Retrieve operation for a sorted list *that can contain duplicate keys*.

```

procedure Retrieve (List : in List_Type; -- List to search
 Key : in Key_Type; -- Key to look for
 Elements : in out List_Type); -- A list of all
-- elements with
-- the given Key

```

```

-- Retrieve all of the elements with the given Key.
--
-- Preconditions : None
--
-- Postconditions : Elements contains copies of all elements in List
-- with the given Key. If no element in List
-- has the Key, the list Elements will be empty.

```

The list is implemented as a singly linked list with header and trailer nodes. Write the procedure body for this operation. You may call the search procedure whose specification was given in Exercise 17. One of the first thing you should do is call procedure `Clear` for list `Elements`.

20. Why did we implement the unbounded natural number class described in the case study of this chapter as a limited private type rather than a private type?

21. Trace the following application program that uses a stack of unbounded natural numbers. Draw pictures showing the stack and the variable `Number` after the first, second, and third iterations of the for loop. This exercise illustrates the importance of separating the different levels of abstraction used in a program. Do not consider how the unbounded natural class is implemented. Treat objects of this class just like any integer type.

```

with Stack;
with Text_IO;
with Unbounded_Natural;
procedure Stack_Example is

 -- A type that designates an unbounded number
 type Ptr_Type is access Unbounded_Natural.Natural;
 -- A package for stacks of pointers
 package Num_Stack is new Stack (Element_Type => Ptr_Type);

 Number : Ptr_Type; -- A pointer to an unbounded natural number
 Stack : Num_Stack.Stack_Type (Max_Size => 20); -- A stack of pointers

begin -- Program Stack_Example
 Text_IO.Put_Line ("Enter 3 numbers");
 for I in 1..3 loop
 -- Get memory for an unbounded natural number
 Number := new Unbounded_Natural.Natural;
 -- Get the number from the keyboard
 Unbounded_Natural.Get (Item => Number.all);
 -- Push a pointer (an alias) to the number onto the stack
 Num_Stack.Push (Stack => Stack, New_Element => Number);
 end loop;
 Text_IO.Put_Line ("Here they are in reverse order");
 loop
 exit when Num_Stack.Empty (Stack);
 Num_Stack.Pop (Stack => Stack, Popped_Element => Number);
 Unbounded_Natural.Put (Item => Number.all, Width => 10);
 Unbounded_Natural.Finalize (Number)
 Text_IO.New_Line;
 end loop;
end Stack_Example;

```



## Programming Problems

1. Implement and test the sorted list ADT body using only a trailer node (no header node). You must modify the declaration of `List_Type` in Specification 8-1. Turn in the following:

- (a) written test plan
- (b) listing of your list package declaration
- (c) listing of your list package body
- (d) listing of your test program
- (e) output from your test program

2. Implement and test the sorted list ADT body using only a header node (no trailer node). You must modify the declaration of `List_Type` in Specification 8-1. Turn in the following:

- (a) written test plan
- (b) listing of your list package declaration
- (c) listing of your list package body
- (d) listing of your test program
- (e) output from your test program

3. Extend the List ADT with the operations described in Chapter 8 programming problem 2 using a list with both header and trailer nodes. Turn in the following:

- (a) written test plan for the new operations
- (b) listing of your list package declaration
- (c) listing of your list package body
- (d) listing of your test program
- (e) output from your test program

4. (a) Add the following operations to the specification of the unbounded natural number class (Specification 9-4) and add their implementation to Body 9-1:

```

procedure Convert_To_Unbounded (Value : in Standard.Natural;
 Result : out Natural);
-- Convert a standard natural number to an unbounded natural number

procedure Convert_To_Standard (Value : in Natural;
 Result : out Standard.Natural);
-- Convert an unbounded natural number to an standard natural number
-- CONSTRAINT_ERROR is raised if Value > Standard.Natural'Last

function "<" (Left : in Natural; Right : in Natural) return Boolean;
-- Returns True if the left unbounded natural number is less than
-- the right unbounded natural number

function "<=" (Left : in Natural; Right : in Natural)
 return Boolean;
-- Returns True if the left unbounded natural number is less than
-- or equal to the right unbounded natural number

function ">" (Left : in Natural; Right : in Natural) return Boolean;
-- Returns True if the left unbounded natural number is greater than
-- the right unbounded natural number

function ">=" (Left : in Natural; Right : in Natural)
 return Boolean;
-- Returns True if the left unbounded natural number is greater than
-- or equal to the right unbounded natural number

```

```
function "=" (Left : in Natural; Right : in Natural) return Boolean;
-- Returns True if the two unbounded natural numbers are equal
```

(b) Why can we write functions for the relational operations with unbounded natural numbers as functions, but must write procedures for the arithmetic operations?

(c) Write a test plan for the operations defined in part (a).

(d) Write a program to implement your test plan

5. The algorithm used in Body 9-1 to divide two unbounded natural numbers is very inefficient. Implement a more efficient algorithm for dividing unbounded natural numbers based on the technique you first learned for long division.

```

 0 0 0 7 2
451) 3 8 9 7 1
 3 6 0 8
 1 8 9 1
 1 3 5 3

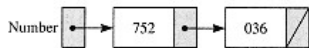
 4 3 8

```

6. Implement an unbounded integer class. Use a circular doubly linked list with a single dummy node containing -1 if the number is negative and +1 if it is positive. Write a program to test your ADT.

7. Our implementation for an unbounded natural number might use more space for the pointers than for the digits. To save space, each node can contain several digits, as illustrated by the storage of 752,036 in the following figure.

Number = 752,036



Modify the private part of Specification 9-4 and Body 9-1 so that three digits are stored in every node instead of one digit. Write a program to test your modified implementation.

8. Write a program to implement a four-function (+, -, \* and /) postfix calculator for unbounded natural numbers using the unbounded natural class developed in this chapter. Each number and each operator is entered on a separate line. When an = is entered, display the answer. Here is a sample run of the calculator program:

```

1234
2221
+
5265
3221
-
*
=
7062020

```

We developed a similar program in Chapter 6 as an application of stacks. Because the unbounded class is *limited* private, it cannot be used as a generic actual parameter to the stack package developed in Chapter 6. There are two ways to overcome this difficulty. First, you can rewrite the stack with a limited private generic formal parameter. This solution requires that you add a generic formal subprogram for assignment which is used in the stack package body. Another solution is illustrated in Exercise 21; use a stack of pointers to unbounded natural numbers.

9. Linked lists are often used to implement sparse tables. A *sparse table* is a table with relatively few nonzero elements. Consider a table of company sales figures in which the rows represent the different salespeople and the columns represent the various products sold by the company. Each element in the table contains the total quantity of a particular product that has been sold by one of the salespeople. If the various salespeople specialize, each selling only a small range of different products, it is likely that many or most of the numbers in the table will be zero (see Figure 9-21). This is an example of a sparse table:

|        |  | Product number (columns) |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |  |
|--------|--|--------------------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|--|
|        |  | 003                      | 004 | 026 | 056 | 072 | 124 | 155 | 237 | 274 | 287 | ... | 822 | 853 | 933 | 945 | 949 |  |
| Addams |  | 0                        | 36  | 91  | 0   | 0   | 0   | 0   | 0   | 0   | 28  | ... | 0   | 0   | 0   | 0   | 0   |  |
| Baker  |  | 93                       | 0   | 0   | 33  | 59  | 0   | 0   | 0   | 0   | 0   | ... | 0   | 0   | 56  | 0   | 0   |  |
| Cole   |  | 39                       | 0   | 0   | 26  | 55  | 0   | 0   | 0   | 0   | 33  | ... | 0   | 0   | 39  | 0   | 5   |  |
| Dale   |  | 0                        | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | ... | 0   | 76  | 47  | 98  | 45  |  |
| .      |  |                          |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |  |
| .      |  |                          |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |  |
| .      |  |                          |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |  |
| Xavier |  | 0                        | 20  | 23  | 33  | 64  | 0   | 0   | 0   | 0   | 0   | ... | 36  | 0   | 0   | 0   | 0   |  |
| Young  |  | 0                        | 0   | 0   | 0   | 0   | 54  | 46  | 78  | 36  | 0   | ... | 71  | 0   | 0   | 0   | 0   |  |
| Zorro  |  | 48                       | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 69  | ... | 0   | 87  | 0   | 67  | 0   |  |

Figure 9-21  
A Sparse Table

Though it is natural to think of implementing a table as a two-dimensional array, a sparse table may be more efficiently implemented (with regard to space) as a linked list structure. One way to represent a sparse table with linked lists is pictured in Figure 9-22. The rows represent the different salespeople and the columns represent the different product numbers. Each row and column has a header node. Unlike the header nodes we used earlier in this chapter, these header nodes contain information to identify a particular row or column.

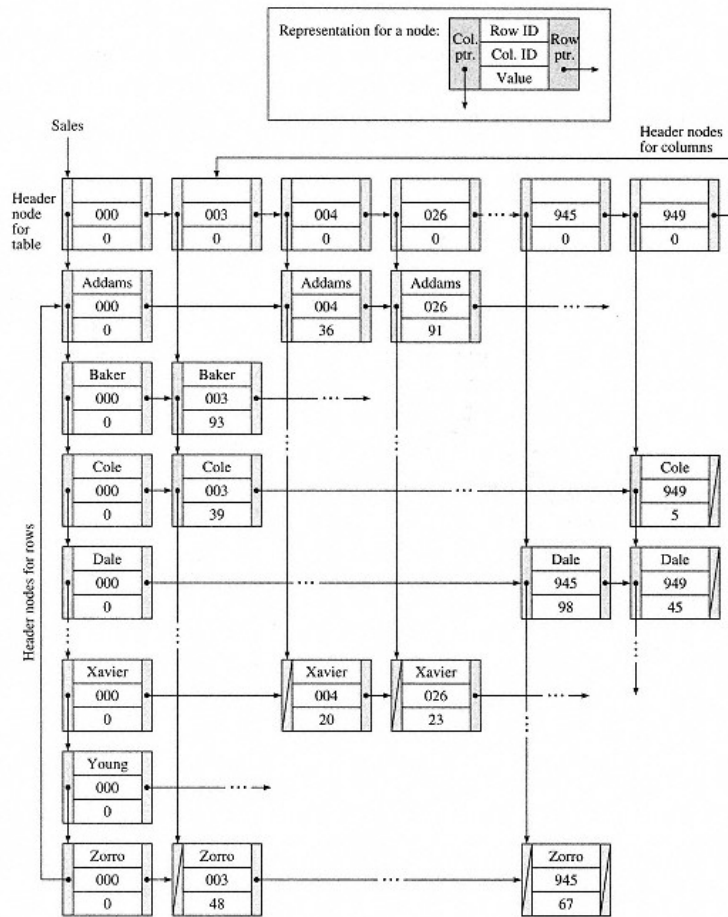


Figure 9-22  
A Sparse Table Represented as Linked Lists with Header Nodes

Here is a specification of a sparse array class with discrete index types and an integer component type:

```
generic
 type Row_Index is (<>);
 type Column_Index is (<>);
 type Value_Type is range <>;
package Sparse_Array is

 type Array_Type is limited private;

 procedure Update (The_Array : in out Array_Type;
 Row : in Row_Index;
 Column : in Column_Index;
 Value : in Value_Type);
 -- Equivalent to The_Array (Row, Column) := Value;
 -- Values of zero use no memory

 function Value_Of The_Array : in out Array_Type;
 Row : in Row_Index;
 Column : in Column_Index)
 return Value_Type;
 -- Returns the value of The_Array (Row, Column)

private

 type Node_Type;

 type Array_Type is access Node_Type;
 subtype Node_Ptr is Array_Type; -- A synonym

 type Node_Type is
 record
 Row_ID : Row_Index;
 Col_ID : Column_Index;
 Value : Value_Type;
 Col_Ptr : Node_Ptr;
 Row_Ptr : Node_Ptr;
 end record;

end Sparse_Array;
```

Only nonzero values are stored in this linked structure. Function `Retrieve` searches the data structure for a node containing the given row and column values. If it finds the node, its value is returned; otherwise, zero is returned. Procedure `Update` also searches for the node containing the given row and column values. If it finds the node, it updates the value of the node. If the value is zero, it removes the node from the linked structure. If it does not find the node and `Value` is not zero, it inserts a new node with the given row and column indexes and the given value.

Using the linked structure illustrated in Figure 9-22, implement the body of this sparse array class.

### ***Turn In***

- (a) written test plan for the two operations
- (b) listing of your sparse array package body
- (c) listing of your test program
- (d) output from your test program

## **10— Alternative Storage of Linked Structures**

### **GOALS**

After reading this chapter, you should be able to

- implement a linked list in an array of records
- implement a linked list in a direct file

We used dynamic memory allocation to obtain the necessary memory for the nodes making up the linked structures developed in previous chapters. There are times when dynamic allocation isn't possible or feasible, and when a linked list must be kept in a static structure. In this chapter we examine two alternative ways to implement linked structures.

### The Linked List As an Array of Records

We mentioned briefly in Chapter 8 that the values in a linked list are not necessarily stored in dynamically allocated memory. They can also be stored in a *static* structure, the array. Let's see how this implementation might work. The elements in the list can be stored in an array of records, with each node in the list consisting of a record with two fields, `Info` and `Next`. Unlike the sequential list implementation developed in Chapter 8, these records can be stored *in any physical order* in the array. Their logical order is specified explicitly through the value in their `Next` fields. The `Next` field should point out which node (array slot) comes next in the list. We don't need to use an Ada access type for this; we only need to know the index ("address" in the array) of the succeeding list element.

#### Why Use an Array?

We have seen that dynamic allocation has many advantages, so why would we even discuss using a static implementation instead? We have discussed that dynamic allocation is only one issue in choosing a linked implementation; another advantage is the efficiency of the insert and delete algorithms. Most of the algorithms that we have discussed for operations on a linked list can be used for either a static or a dynamic implementation. The main difference is the requirement that we manage our own free space in an array-based implementation. Managing the free space ourselves gives us greater flexibility.

Another reason to use an array is that a number of programming languages do not have dynamic allocation or access types. You can still use linked lists if you are programming in one of these languages, but you would have to represent access values as array indexes.

Using access variables presents a problem when we need to save the information in a data structure between runs of a program. Suppose we want to write all the nodes in a list to a file and then use this file as input the next time we run the program. If the links are access values—containing memory addresses—they are meaningless on the next run of the program. We must save the user data part of each node in the file, and then rebuild the linked structure the next time we run the program (like we did in the Address Book application in Chapter 8). An array index, however, is still valid on the next run of the program. We can store the whole array, including the `Next` fields (indexes), and then read it back in the next time we run the program.

#### How Is an Array Used?

Let's get back to our discussion of how a linked list can be implemented in an array. As we have said, the `Next` field of each node tells us the array index of the succeeding node. The beginning of the list is accessed through a "pointer" that contains the array index of the first element in the list. Figure 10-1 shows how an ordered list containing the elements David, Joshua, Leah, Miriam, and Robert might be stored in an array of records called `Nodes`. Do you see how the order of the elements in the list is explicitly indicated by the chain of `Next` indexes?

What goes in the `Next` field of the last list element? Its "null" value must be an invalid address for a real list element. Because the `Nodes` array indexes begin at 1, the value 0 is not a valid index into the array; that is, there is no `Nodes ( 0 )` . Therefore 0 makes an ideal value to use as a "null" address. We could use the literal value 0 in our programs

```
loop
 exit when Location = 0
 .
 .
 .
```

but it is better programming style to declare a constant. We can't use the identifier `null`, however, since this is an Ada reserved word. We'll use the identifier `Nil` instead:

```
Nil : constant := 0;
```

When an array of records implementation is used to represent a linked list, the programmer must write routines to manage the free space available for new list elements.

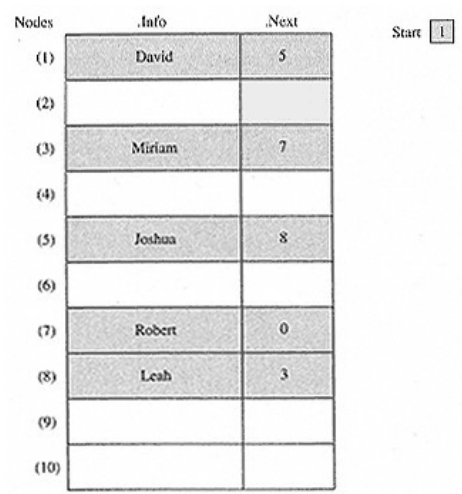


Figure 10-1  
An Ordered List Stored in an Array

Where is this free space? Look again at Figure 10-1. All of the array elements that do not contain values in the list constitute free space. Instead of the built-in allocator `new`, which allocates memory dynamically, we must write our own function to allocate nodes from the free space. We call this function `New_Node`. To be consistent with Ada's `new` allocator, we can write two versions of `New_Node`; one where an initial value for the node is supplied, and one where it is not.



When elements are deleted from the list, we need to free the node space. We can't use `Unchecked_Deallocation` because it only works for dynamically allocated space. We write our own procedure, `Free_Node`, to put a node back in the pool of free space.

This collection of unused array elements can be linked together into a second list, a linked list of free nodes. Figure 10-2 shows the array `Nodes` with both the list of values and the list of free space linked through their `Next` fields. The list of values begins at `Start`, at index 1 (containing the value David). Following the links in the `Next` field, we see that the list continues with the array slots at index 5 (Joshua), 8 (Leah), 3 (Miriam), and 7 (Robert), in that order. The free list begins at `Free`, at index 2. Following the links in the `Next` field, we see that the free list also includes the array slots at index 6, 4, 9, and 10. You see two `Nil` values in the `Next` field column because there are two linked lists contained in the `Nodes` array. We can use one array to store many different linked lists. Figure 10-3 shows an array that contains two different lists. `List 1` contains the values John, Nell, Susan, Suzanne and `List 2` contains the values David, Naomi, and Robert. The remaining three array slots in Figure 10-3 are linked together in the free list.

Now let's look at how the list is declared. In the variation of Specification 8-1 on page 441, we have added the generic formal object `Max_Size` so that the user may set the size of our storage array. `Max_Size` determines the combined number of elements that can be stored in *all* the list objects declared from this package. Having to specify this maximum size is a disadvantage of the array-based linked list not found in the access-based linked list. In the private part we again enclose the external pointer in a record to allow us to assign an initial value of "empty" to every list object declared.

| Nodes | Info    | Next | Start | Free |
|-------|---------|------|-------|------|
| (1)   | David   | 5    | 1     | 2    |
| (2)   | Garbage | 6    |       |      |
| (3)   | Miriam  | 7    |       |      |
| (4)   | Garbage | 9    |       |      |
| (5)   | Joshua  | 8    |       |      |
| (6)   | Garbage | 4    |       |      |
| (7)   | Robert  | 0    |       |      |
| (8)   | Leah    | 3    |       |      |
| (9)   | Garbage | 10   |       |      |
| (10)  | Garbage | 0    |       |      |

Figure 10-2  
An Array with Linked Lists of Values and Free Space

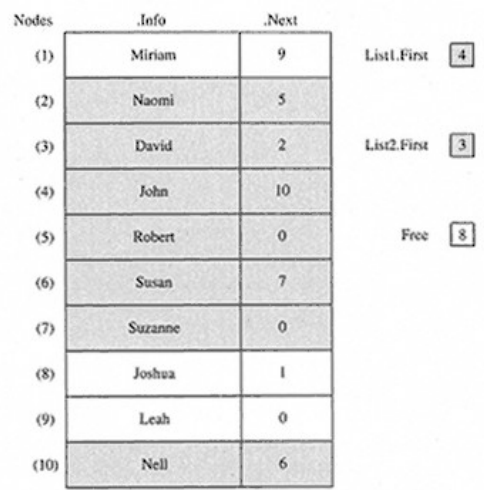


Figure 10-3  
An Array Containing Two Linked Lists of Values and a Linked  
List of Free Nodes

```
generic

Max_Size : in Positive; -- The storage for ALL list objects

type Element_Type is private; -- The type of element in the list
type Key_Type is limited private; -- The type of key in the element

-- The user must supply a function that returns the Key of an Element.
with function Key_Of (Element : in Element_Type) return Key_Type;
-- The user must supply functions for comparing Keys.
with function "=" (Left : in Key_Type; Right : in Key_Type)
 return Boolean;
with function "<" (Left : in Key_Type; Right : in Key_Type)
 return Boolean;

package Sorted_List is

 -- Same list operations as before (Specification 8-1)

private -- Linked List stored in an array

 subtype Node_Ptr is Integer range 0..Max_Size; -- "Pointer" to a node

 Nil : constant Node_Ptr := 0;
```

```

type List_Type is
 record
 First : Node_Ptr := Nil; --Initialize list to empty
 end record;

end Sorted_List;

```

Most of the types for this implementation are completely hidden in the package body. The body also contains variables for the storage array `Nodes` and a pointer, `Free`, to the first node in the list of free nodes. These variables are accessed globally by the procedures in the package body. But because they are declared within the body, they are not available outside of the package body. Here are the types and variables we declare in our package body:

```

package body Sorted_List is

 -- Implements a linked list stored in an array.

 subtype Index_Range is Node_Ptr range 1..Node_Ptr'Last;

 type Node_Type is
 record
 Info : Element_Type;
 Next : Node_Ptr := Nil;
 end record;

 type Node_Array is array (Index_Range) of Node_Type;

 -- Package Variables (used globally in the package)

 Nodes : Node_Array; -- Storage array for the nodes
 Free : Node_Ptr; -- Head of list of available nodes
 .
 .
 .

```

Converting the code used to implement the operations for an access-based linked list to an array-based linked list only takes a little rearranging. With the array implementation, nodes are accessed through a pointer that is an array index rather than an access type. Compare the following two assignment statements to copy the information in the node to which `Location` "points":

```

Element := Location.all.Info; -- Access-based linked list
Element := Nodes(Location).Info; -- Array-based linked list

```

In the access-based linked list, the pointer is an access value that designates a node somewhere in memory. In the array-based linked list, the pointer is an index that selects a node somewhere in the array. Figure 10-4 shows how the two implementations correspond.

To show you how easy it is to change the implementation, let's look at the code for the insert procedure. First, here is the code for the access-based linked list:

```
-- Insert into an access-based linked list
Search_Linked_List (List => List,
 Key => Key_Of (Item),
 Found => Have_Duplicate,
 Pred_Loc => Pred_Loc,
 Location => Location);

if Have_Duplicate then
 raise DUPLICATE_KEY;
elsif Pred_Loc = null then -- Add at beginning?
 List := new Node_Type'(Info => Item, Next => List);
else -- Add at the middle or end
 Pred_Loc.all.Next := new Node_Type'(Info => Item, Next => Location);
end if;
```

And here is the "same" code for an array-based linked list:

```
-- Insert into an array-based linked list
Search_Linked_List (List => List,
 Key => Key_Of (Item),
 Found => Have_Duplicate,
 Pred_Loc => Pred_Loc,
 Location => Location);

if Have_Duplicate then
 raise DUPLICATE_KEY;
elsif Pred_Loc = Nil then -- Add at beginning?
 List.First := New_Node ((Info => Item, Next => List.First));
else -- Add at the middle or end
 Nodes(Pred_Loc).Next := New_Node ((Info => Item, Next => Location));
end if;
```

| Algorithm                  | Array of Records                     | Dynamic Storage            |
|----------------------------|--------------------------------------|----------------------------|
| Allocate a node            | New_Node (Initial_Value)<br>New_Node | new'(Initial_Value)<br>new |
| Free a node                | Free_Node (Ptr)                      | Free (Ptr)                 |
| Access Info                | Nodes(Ptr).Info                      | Ptr.all.Info               |
| Access Next                | Nodes(Ptr).Next                      | Ptr.all.Next               |
| Exit when no more elements | exit when Ptr = Nil                  | exit when Ptr = null       |

Figure 10-4  
Two Implementations of Linked Lists

Changing the linked list operations that we coded in Chapter 8, this array-based implementation is so simple that much of the work to make this conversion can be accomplished through "search and replace" commands available in a program editor. That is exactly what we did to produce the implementation given in Body 10-1.

**Body 10-1—****Array-Based Linked List Implements Specification 8-1**

```

package body Sorted_List is

-- Implements a linked list stored in an array.

subtype Index_Range is Node_Ptr range 1..Node_Ptr'Last;

type Node_Type is
 record
 Info : Element_Type;
 Next : Node_Ptr := Nil;
 end record;

type Node_Array is array (Index_Range) of Node_Type;
-- Package Variables (used globally in the package)

Nodes : Node_Array; -- Storage array for the nodes
Free : Node_Ptr; -- Head of list of available nodes

-- Local procedures

procedure Initialize_Memory is
-- Initializes memory management for the array of nodes.
begin
 Free := Nodes'First;
 -- Link all nodes together into the free list
 for Index in Nodes'First .. Nodes'Last - 1 loop
 Nodes(Index).Next := Index + 1;
 end loop;
 Nodes(Nodes'Last).Next := Nil; -- Mark the end of free list
end Initialize_Memory;

function New_Node (Initial_Value : in Node_Type) return Node_Ptr is
-- Finds a free node in the array (memory allocation).
 Result : Node_Ptr;
begin
 if Free = Nil then
 raise OVERFLOW;
 else -- Get a node from the free list
 Result := Free;
 Free := Nodes(Free).Next; -- Unlink the free node
 Nodes(Result) := Initial_Value; -- Assign the initial value
 return Result;
 end if;
end New_Node;

```

```

procedure Free_Node (X : in out Node_Ptr) is
-- Deallocates a node.
begin
 Nodes(X).Next := Free; -- Link the node
 Free := X; -- into the free list
 X := Nil; -- Set the user's pointer to Nil
end Free_Node;

procedure Search_Linked_List (List : in List_Type;
 Key : in Key_Type;
 Found : out Boolean;
 Pred_Loc : out Node_Ptr;
 Location : out Node_Ptr) is
-- Searches for the location of Key in the List.

-- Preconditions : The nodes in List are in ascending order.

-- Postconditions : If Key is in List
-- Found is True.
-- Location designates the element with Key.
-- Pred_Loc designates the predecessor of the node designated
-- by Location. If Location designates the first
-- node in the list, Pred_Loc is null.
-- else
-- Found is False.
-- Location designates the node that would follow a node
-- containing Key. If Key's node would follow
-- the last node in the list, Location is null.
-- Pred_Loc designates the predecessor of the node designated
-- by Location. If Location designates the first
-- first node in the list, Pred_Loc is null.

 Current : Node_Ptr; -- Local copy of Location
 Previous : Node_Ptr; -- Local copy of Pred_Loc
begin
 Current := List.First; -- Start at the beginning of the linked list
 Previous := Nil; -- No predecessor for 1st element in the list
 loop -- Each iteration, one node is checked
 exit when Current = Nil or else -- Reached the end
 Key < Key_Of (Nodes(Current).Info) or else -- Passed by
 Key = Key_Of (Nodes(Current).Info); -- Found it
 Previous := Current; -- Move inchworm's tail
 Current := Nodes(Current).Next; -- Move inchworm's head
 end loop;
 -- Copy results to out parameters
 Location := Current;
 Pred_Loc := Previous;
 Found := Current /= Nil and then Key = Key_Of (Nodes(Current).Info);
end Search_Linked_List;

-- Operations

```

```

procedure Clear (List : in out List_Type) is
 To_Recycle : Node_Ptr; -- Designates a node for recycling.
begin
 loop
 exit when List.First = Nil; -- Exit when list is empty
 To_Recycle := List.First; -- Unlink the
 List.First := Nodes(List.First).Next; -- the first node
 Free_Node (To_Recycle); -- Recycle the node
 end loop;
end Clear;

```

```

procedure Insert (List : in out List_Type;
 Item : in Element_Type) is
 Have_Duplicate : Boolean;
 Pred_Loc : Node_Ptr;
 Location : Node_Ptr;
begin
 Search_Linked_List (List => List,
 Key => Key_Of (Item),
 Found => Have_Duplicate,
 Pred_Loc => Pred_Loc,
 Location => Location);

 if Have_Duplicate then
 raise DUPLICATE_KEY;
 elsif Pred_Loc = Nil then -- Add at beginning
 List.First := New_Node ((Info => Item, Next => List.First));
 else -- Add at the middle or end
 Nodes(Pred_Loc).Next := New_Node ((Info => Item, Next => Location));
 end if;
end Insert;

```

```

procedure Delete (List : in out List_Type;
 Key : in Key_Type) is
 Found : Boolean;
 Pred_Loc : Node_Ptr;
 Location : Node_Ptr;
begin
 Search_Linked_List (List => List,
 Key => Key,
 Found => Found,
 Pred_Loc => Pred_Loc,
 Location => Location);

 if not Found then
 raise KEY_ERROR;
 elsif Pred_Loc = Nil then
 List.First := Nodes(Location).Next; -- Delete first element
 else
 Nodes(Pred_Loc).Next := Nodes(Location).Next; -- Delete middle or last element
 end if;
 Free_Node (Location); -- Recycle node memory
end Delete;

```

```

procedure Modify (List : in out List_Type;
 Element : in Element_Type) is
```

```

 Found : Boolean;
 Pred_Loc : Node_Ptr;
 Location : Node_Ptr;
begin
 Search_Linked_List (List => List,
 Key => Key_Of (Element),
 Found => Found,
 Pred_Loc => Pred_Loc,
 Location => Location);

 if not Found then
 raise KEY_ERROR;
 else
 Nodes(Location).Info := Element;
 end if;
end Modify;
```

```

procedure Retrieve (List : in List_Type;
 Key : in Key_Type;
 Element : out Element_Type) is
```

```

 Found : Boolean;
 Pred_Loc : Node_Ptr;
 Location : Node_Ptr;
begin
 Search_Linked_List (List => List,
 Key => Key,
 Found => Found,
 Pred_Loc => Pred_Loc,
 Location => Location);

 if not Found then
 raise KEY_ERROR;
 else
 Element := Nodes(Location).Info;
 end if;
end Retrieve;
```

```

function Full (List : in List_Type) return Boolean is
begin
 return Free = Nil;
end Full;
```

```

function Empty (List : in List_Type) return Boolean is
begin
 return List.First = Nil;
end Empty;
```



```

function Length (List : in List_Type) return Natural is
 Location : Node_Ptr; -- Designates current node
 Count : Natural;
begin
 Count := 0;
 Location := List.First;
 loop
 exit when Location = Nil;
 Count := Count + 1;
 Location := Nodes(Location).Next;
 end loop;
 return Count;
end Length;

procedure Traverse (List : in out List_Type) is
 Element : Element_Type; -- Use local variable to ensure keys are not changed
 Location : Node_Ptr; -- Designates current node
begin
 Location := List.First;
 loop
 exit when Location = Nil;
 Element := Nodes(Location).Info; -- Make a local copy of the element
 Process (Element); -- Call user's procedure to process it
 -- Make sure the Key wasn't changed
 if Key_Of (Element) /= Key_Of (Nodes(Location).Info) then
 raise KEY_ERROR;
 else -- Put possibly modified element back into list
 Nodes(Location).Info := Element;
 end if;
 Location := Nodes(Location).Next; -- Move to next node
 end loop;
end Traverse;

begin -- Package Initialization

 Initialize_Memory; -- Link all of the nodes into the free list

end Sorted_List;

```

### ***Array Memory Management***

Body 10-1 contains three additional local operations. These operations are necessary to manage the memory in the array in which all nodes are stored. We said earlier that we need to do our own memory management by writing the subprograms `New_Node` and `Free_Node`. The code for procedure `Insert` illustrates a call to function `New_Node`.

`New_Node` returns a node pointer (the index of an entry in the array `Nodes`). The version of `New_Node` included in Body 10-1 has a parameter that is assigned as the initial value to the newly acquired node. Using our box-and-arrow pictures, let's see how the `New_Node` operation works. Figure 10-5 shows two linked lists, one containing the user's data (accessed through `List.First`) and the other containing nodes that are not in use (accessed through `Free`).

As calls are made to `New_Node`, nodes are removed from the list of free nodes. Because the free list doesn't contain "real" data and isn't ordered in any way, we can take any node from it. The simplest approach is just to take the first element from the free list. Figure 10-6 illustrates how the function `New_Node` affects the free list. Figure 10-6(a) shows that `Result` is set to designate the first node in the free list. Then, as illustrated in Figure 10-6(b), this first node is removed from the free list. The algorithm for function `New_Node` is nearly identical to the algorithm we developed in Chapter 6 for the linked version of the stack procedure `Pop`. In fact, we could just consider the free "list" to be a linked *free-space stack*.

To better duplicate Ada's `new` operator for access types, we make one addition to the `pop` algorithm. We include a parameter containing the initial value we wish to assign to the newly obtained node. Body 10-1 contains the code for procedure `New_Node`.

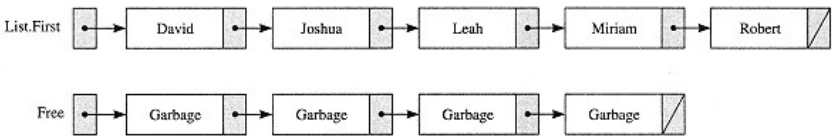


Figure 10-5  
The Two Lists

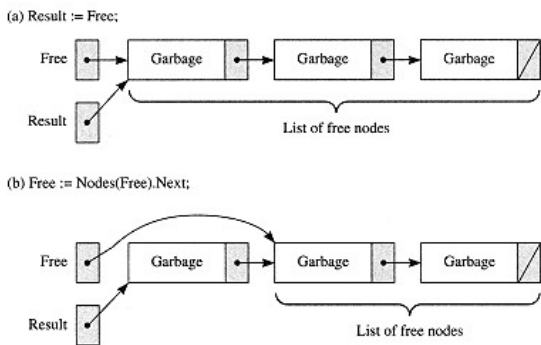


Figure 10-6  
The `New_Node` Operation

Calls to `Free_Node` move nodes back into the list of available nodes. We could add them anywhere in the free list, but again the simplest approach is just to insert them at the beginning of the list as if it were a stack. Figure 10-7 illustrates how the `Free_Node`

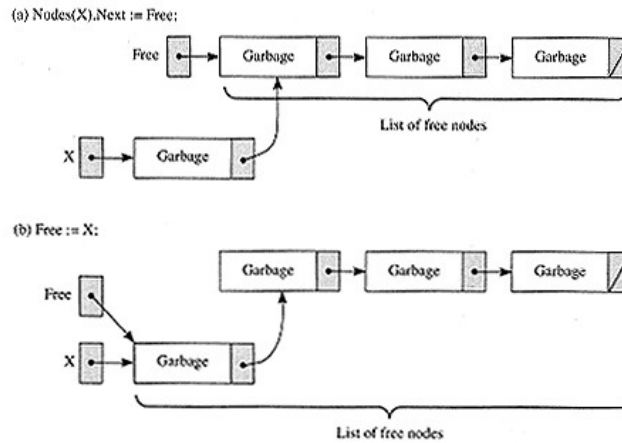


Figure 10-7  
The `Free_Node` Operation

operation affects the free list. In Figure 10-7(a), the recycled node is linked into the free list by setting its `Next` field to designate the current first node in the free list. The linking is completed in Figure 10-7(b) where `Free` is changed to make our recycled node the first node in the free list. Body 10-1 contains the code for function `Free_Node` (using the parameter name `X` to be consistent with the parameter name used by Ada's `Unchecked_Deallocation` procedure). This operation is just like the linked version of procedure `Push`, with `X` as the pointer to the node being returned to the free list.

But we're not finished. How did the nodes get into the free list in the first place? All of the nodes in the array should start out linked together in the list of free-space nodes, which is accessed through the pointer field, `Free`. The free nodes can be linked together in any order, of course, but it is simplest to string them together sequentially, as illustrated in Figure 10-8. Procedure `Initialize_Memory` in Body 10-1 links all of the nodes together in this sequential fashion.

We include a call to procedure `Initialize_Memory` at the end of our package body so that it is called whenever our package is elaborated. We first took advantage of this package initialization feature in the Address Book application in Chapter 8.

One final word of advice. When developing algorithms for array-based linked lists, use the box-and-arrow pictures that we always draw when working with linked lists (Figure 10-5) instead of array pictures (Figure 10-2). The box-and-arrow pictures include only the information necessary to manipulate the list. The unimportant details, such as where the nodes are physically stored, aren't shown. As usual, choosing the correct level of abstraction makes problem solving easier.

### *Analysis of the Array-Based List*

Because the algorithms for the array-based linked list are identical to those developed for linked lists stored in dynamic memory, the Big-O and number of source lines of code are

| Nodes | Info    | .Next | List.First | Free |
|-------|---------|-------|------------|------|
| (1)   | Garbage | 2     | 0          | 1    |
| (2)   | Garbage | 3     |            |      |
| (3)   | Garbage | 4     |            |      |
| (4)   | Garbage | 5     |            |      |
| (5)   | Garbage | 6     |            |      |
| (6)   | Garbage | 7     |            |      |
| (7)   | Garbage | 8     |            |      |
| (8)   | Garbage | 9     |            |      |
| (9)   | Garbage | 10    |            |      |
| (10)  | Garbage | 0     |            |      |

Figure 10-8  
Initializing the Free List to Contain All the Nodes in the Array

also identical. Of course, in the array-based implementation there are three additional procedures for managing the array memory. We leave the determination of the Big-O of these procedures as an exercise.

Let's see what our experiments (described in Chapter 8) can tell us about the array-based implementation. Each number in the following table is the number of seconds required to insert or delete 1000 elements.

**Results of the Timed Implementation Comparisons (in seconds)**

|                           | <i>Linked<br/>List<br/>(Chapter 8)</i> | <i>Array-Based<br/>Linked<br/>List</i> | <i>Sequential<br/>Array-Based<br/>List (Chapter 8)</i> |
|---------------------------|----------------------------------------|----------------------------------------|--------------------------------------------------------|
| <b>Case 1: Random</b>     |                                        |                                        |                                                        |
| Insert                    | 0.71                                   | 0.85                                   | 1.98                                                   |
| Delete                    | 0.67                                   | 0.86                                   | 1.95                                                   |
| <b>Case 2: List End</b>   |                                        |                                        |                                                        |
| Insert                    | 1.27                                   | 1.69                                   | 1.64                                                   |
| Delete                    | 1.28                                   | 1.68                                   | 1.57                                                   |
| <b>Case 3: List Start</b> |                                        |                                        |                                                        |
| Insert                    | 0.03                                   | 0.02                                   | 2.46                                                   |
| Delete                    | 0.02                                   | 0.01                                   | 2.46                                                   |

The experiments indicate that for this specific computer, the array-based linked list is slightly slower than a linked list implemented with dynamic memory. This makes sense.

We have used the exact same algorithms, but now the program must also calculate addresses from array indexes using the formulas developed in Chapter 3. However, except for inserting and deleting at the end, the times recorded for the array-based *linked list* are significantly better than those for the *sequential* array-based list.

### The Linked List As a File of Records

We have stored the information for all of our data structures in memory, either using a static array or dynamic memory. There are times, however, when the amount of data is too large to fit in the computer's memory. For such applications, we must store the information in secondary memory, usually on magnetic disks. Keeping our list information on disk also means that it will be there the next time we run our program. Using Ada's `Direct_IO` package, modifying any of our list algorithms to use disk memory instead of RAM isn't difficult.

#### Direct Files

---

##### Direct File (Random Access File)

A binary file whose components can be accessed sequentially or randomly (in any order).

##### Sequential File

A binary file whose components can be accessed sequentially.

##### Binary File

A file data type whose components are stored in the internal binary representation of the machine.

---

**Direct files**, sometimes called **random access files**, are binary files. **Sequential files**, like the one used in the Address Book application in Chapter 8, are also binary files. A **binary file** is a file whose components are stored using the internal representation of the machine. The components of a binary file are not restricted to characters as a text file is. Because no data conversions are made, input and output with binary files is usually much faster than input and output with text files.

Figure 10-9 illustrates a direct file. A direct file is similar to an array (compare Figures 10-1 and 10-9). We can access any component in a direct file by specifying a component number much like we can access any array component by specifying an index. Like the array, the direct file is one of Ada's predefined classes. Because the direct file class is defined in the generic package `Direct_IO`, this class is used just like those we define; package `Direct_IO` contains a `File_Type` and functions and procedures for the operations on that type. Appendix E gives the declaration of package `Direct_IO`. Many of the operations, like `Open`, `Create`, and `Close`, are familiar from our work with text files. We need three other operations to implement a sorted list in a direct file. Procedure `Read` moves a copy of a file component into a variable. Procedure `Write` moves a copy of a variable into a file component. And function `Size` returns the number of components in the file. Here are the declarations of these three operations and some associated types:

```
-- Declarations found in Package Direct_IO

type File_Type is limited private; -- The direct file type

type Count is range 0..implementation_defined;
subtype Positive_Count is Count range 1..implementation_defined;

procedure Read (File : in File_Type; -- File to read from
 From : in Positive_Count; -- The component number
 Item : out Element_Type); -- Contains copy of component

procedure Write (File : in File_Type; -- File to write to
 To : in Positive_Count; -- Where to write
 Item : in Element_Type); -- The value to write

function Size (File : in File_Type) return Count;
```

Because it is a generic package, we must instantiate a separate package for each different element type for which we need a file. Here are two example instantiations:

```
package Number_IO is new Direct_IO (Element_Type => Integer);
```

```
package Node_IO is new Direct_IO (Element_Type => Node_Type);
```

The first line instantiates a package for a direct file class whose components are integers. The second instantiates a package for a direct file class whose components are nodes. Once we have instantiated a package, we can declare file variables.

```
List_File : Node_IO.File_Type; -- A direct file of Nodes
```

Arrays and files are very similar, but there are some important differences. We discuss two of these differences. First, file element numbers are more restricted than array indexes. The index of an array can be any discrete type and start at any value. File element numbers are always type `Positive_Count` and the first element number in a file is always 1. Second, we do not have to declare a maximum size for a file. While the index of the last component of an array is fixed by its declaration, the last element number in a file increases as elements are written to locations beyond the end of the file. A file may grow in size until there is no room left on the storage device. In this sense, a file is more like the dynamic memory we use to implement linked lists than a static array. Function `Size` returns the element number of the last element in the file.

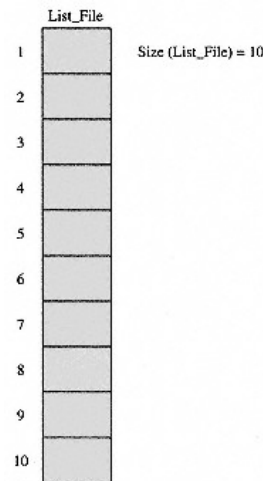


Figure 10-9  
A Direct File Containing 10  
Elements

### *The Abstract Level*

Until now, Specification 8-1 has been adequate for all of the sorted list implementations we have developed. But for a file-based implementation we need some additional information. Each file must have a name compatible with the computer's file system. The programmer using our file-based sorted list class must supply these names; he or she knows the syntax rules for file names on that particular computer and what file names are most appropriate for the application. Each file-based linked list requires two files: One file to store the nodes (equivalent to the array) and another file to store bookkeeping information (including the external pointers for the node list and free list) between runs of the program. At the abstract level, there are two ways to handle the assignment of file names. We could introduce a Create operation that includes file name parameters. We take a second approach. We declare our file-based list as an ADO rather than an ADT (Exercise 7 asks you what changes are necessary to change this package into an ADT). We include generic formal parameters for the file names. The two file names are supplied when the package is instantiated.

Because all files must be closed before our program terminates, we need another "list" operation that is called before the application program terminates. We include a close operation in our specification. Specification 10-1 is the abstract-level view of our file-based linked list. To save space, we have not included the comments for the operations that are identical to those of Specification 8-1.

The generic formal parameters for this list package include two file names. The package uses `List_File_Name` as the external file name of a file in which the list elements are stored and `Header_File_Name` as the external file name of a file in which list bookkeeping information is stored.

Procedure `Close` must be called when the program is finished using the list. Failure to close the list will yield corrupted list files the next time the program is run.

#### **Specification 10-1— File-Based Sorted List ADO**

generic

```
List_File_Name : in String; -- Name of file for list
Header_File_Name : in String; -- Name of file for bookkeeping info

type Element_Type is private; -- The type of element in the list
type Key_Type is limited private; -- The type of key

with function Key_Of (Element : in Element_Type) return Key_Type;
with function "=" (Left : in Key_Type; Right : in Key_Type) return Boolean;
with function "<" (Left : in Key_Type; Right : in Key_Type) return Boolean;
```

package Sorted\_List is

```
-- This package implements a sorted list ADO. The list is physically stored
-- in two files. The user supplies the two file names.
```

```

-- Assumptions : Each element in the list has a unique key.
-- If a list file exists from a previous run, the corresponding
-- header file also exists.

-- The elements in the list are ordered from smallest to largest using the "<"
-- function supplied as a generic parameter. The key value of the predecessor
-- of an element is less than the key value of the element, which is less than
-- the key value of the successor of the element.

DUPLICATE_KEY : exception;
KEY_ERROR : exception;
OVERFLOW : exception;

procedure Clear;

procedure Insert (Item : in Element_Type);

procedure Delete (Key : in Key_Type);

procedure Modify (Element : in Element_Type);

procedure Retrieve (Key : in Key_Type;
 Element : out Element_Type);

function Empty return Boolean;

function Full return Boolean;

function Length return Natural;

generic
 with procedure Process (Element : in out Element_Type);
procedure Traverse;

procedure Close;
-- Saves list header information in a file. Must be called before
-- ending the program or list will be lost.

-- Preconditions : Close was not previously called
--
-- Postconditions : Both list files are closed

end Sorted_List;

```

### ***The Implementation Level***

Because a direct file is so much like an array, it should not be surprising that the file-based list implementation is very similar to the array-based list implementation. The only significant difference is that we use input and output procedures to access individual nodes instead of array indexes. This difference usually results in longer procedures. For example, once the location of a node is known in an array-based list, it only takes a single assignment statement



to modify the `Info` field of that node. As shown in the following procedure, it requires three statements to make the same modification in a file-based list: (1) read a copy of the node from the file, (2) modify the copy, and (3) write the modified copy back to the file.

```

procedure Modify (Element : in Element_Type) is
 Found : Boolean;
 Pred_Loc : Node_Ptr;
 Location : Node_Ptr;
 Node : Node_Type; -- Local copy of node kept on the file
begin
 Search_Linked_List (Key => Key_Of (Element),
 Found => Found,
 Pred_Loc => Pred_Loc,
 Location => Location);

 if not Found then
 raise KEY_ERROR;
 else
 Node_IO.Read (File => List_File, -- Get the Node at Location
 Item => Node,
 From => Node_IO.Positive_Count(Location));
 Node.Info := Element; -- Change the Info field
 Node.IO.Write (File => List_File, -- Put the modified node back
 Item => Node, -- into the file
 To => Node_IO.Positive_Count(Location));

 end if;
end Modify;

```

The `From` and `To` parameters of the `Read` and `Write` procedures are type `Node_IO.Positive_Count`. Procedure `Modify` uses explicit type conversions to convert the variable `Location` to type `Positive_Count`. To understand why these conversions are necessary, we need to look at the file-based list package body (Body 10-2).

In Body 10-2, we declare `Node_Ptr` as a synonym for `Natural`. It would be more accurate if we made `Node_Ptr` a synonym for `Node_IO.Count` because our pointers are actually file element numbers. However, we must declare `Node_Ptr` before we can declare `Node_Type` which we must declare before we instantiate package `Node_IO`. We saw a similar "chicken and egg" problem in Chapter 6. There, we used the incomplete type as a solution. Because an incomplete type may be named only in the definition of an access type, this solution isn't applicable here. Instead we use the two different types and use explicit type conversions to convert between them.

Five package variables are accessed globally by procedures Body 10-2. `List_File` is the direct file in which the list elements *and* free elements are stored. `Count` is the number of elements currently in the list. `List` is a pointer to the first node in our linked list. `Free` is a pointer to the first node in a linked list of free nodes. This list of free nodes is stored in the same file as our list elements; they are intertwined in the file just as they were in the array. Finally, `Header_File` is a sequential file used to store the values of `Count`, `List`, and `Free` between program runs.

The actual reading and writing to the list file is done by two local procedures, `Read_Node` and `Write_Node`. There are two reasons why we chose to write these local procedures instead of calling `Node_IO.Read` and `Node_IO.Write` directly: (1) the complexities of the explicit type conversions are confined to two procedures, and (2) we can more easily add a node-caching scheme to avoid multiple reading of the same node from the file (see programming problem 5).

Memory management is slightly more complicated in the file-base list than it was for the array-based list. The memory we are managing here is that in the disk file rather than in an array. Procedure `Free_Node` uses the same algorithm as the array-based list—the node is added to a list of free nodes. In the array-based list, we initialize the list of free nodes by linking together all of the nodes in the array. Because a file has no declared upper bound, we cannot link *all* of the file nodes together. Instead we initialize the free list to *empty* and use the following algorithm to find an available node on the disk:

#### New\_Node

```

if the list of free nodes is empty then
 return the Size of the list plus 1
else
 return the location of the first node on the free list
end if

```

If there are no nodes on the free list, we extend the size of our file by one element. Nodes are only added to the free list when procedure `Free_Node` is called.

To minimize the number of disk reads (a time-consuming operation), we use the package variable `Count` to keep track of the number of elements in the list. `Count` is returned by function `Length`. `Count` is incremented each time a new element is added to the list and decremented each time an element is deleted from the list. Why can't we call `Node_IO.Size` to determine the number of elements in the list? `Node_IO.Size` returns the total number of nodes in the file. This total includes the nodes in the list *and* the nodes in the free list.

Procedure `Close` closes the direct file used to store the list nodes. This procedure also saves the values of three package variables (`Count`, `List`, and `Free`) that are needed the next time the program is run.

The sequence of statements at the end of the package body executes whenever the package is elaborated. These statements initialize the list. The `List_File` and `Header_File` are opened (using the names supplied by the application programmer as generic objects) and the values of `Count`, `List`, and `Free` are read from `Header_File`. If these files do not yet exist, the open operation raises `NAME_ERROR`. The exception handler for `NAME_ERROR` then creates the `List_File` and initializes `Count`, `List`, and `Free` with values for an empty list.

#### Body 10-2—

##### File-Based Linked List ADO Implements Specification 10-1

```

with Direct_IO;
with Sequential_IO;
package body Sorted_List is

 subtype Node_Ptr is Natural;

 Nil : constant Node_Ptr := 0;

```

```

type Node_Type is
 record
 Info : Element_Type;
 Next : Node_Ptr := Nil;
 end record;

-- For input and output of list nodes
package Node_IO is new Direct_IO (Element_Type => Node_Type);
-- For input and output of bookkeeping information
package Header_IO is new Sequential_IO (Element_Type => Integer);

-- Package Variables (used globally in the package)

Header_File : Header_IO.File_Type; -- Keep header info between runs
List_File : Node_IO.File_Type; -- Storage file for the nodes
Count : Natural := 0; -- Size of the List
List : Node_Ptr := Nil; -- The "head" pointer of the list
Free : Node_Ptr := Nil; -- Head of list of available nodes

-- Local procedures

procedure Read_Node (Location : in Node_Ptr;
 Node : out Node_Type) is
-- Reads a node from the list file
begin
 Node_IO.Read (File => List_File, Item => Node,
 From => Node_IO.Positive_Count(Location));
end Read_Node;

procedure Write_Node (Location : in Node_Ptr;
 Node : in Node_Type) is
-- Writes a node from the list file
begin
 Node_IO.Write (File => List_File, Item => Node,
 To => Node_IO.Positive_Count(Location));
end Write_Node;

function New_Node return Node_Ptr is
-- Finds a free node in the file (memory allocation)
 Result : Node_Ptr;
 Node : Node_Type;
begin
 if Free = Nil then -- Use next place in file
 return Node_Ptr (Node_IO.Size (List_File)) + 1;
 else -- Get a node from the free list
 Result := Free;
 -- Unlink the free node
 Read_Node (Location => Result, Node => Node);
 Free := Node.Next;
 return Result;
 end if;
end New_Node;

```

```

procedure Free_Node (X : in out Node_Ptr) is
 Node : Node_Type;
 -- Deallocates a node
begin
 -- Link the node into the free list
 Node.Next := Free;
 Free := X;
 Write_Node (Location => Free, Node => Node);
 X := Nil; -- Reset pointer
end Free_Node;

procedure Search_Linked_List (Key : in Key_Type;
 Found : out Boolean;
 Pred_Loc : out Node_Ptr;
 Location : out Node_Ptr) is
 -- Searches for the location of Key in the List

 -- Preconditions : The nodes in List are in ascending order.

 -- Postconditions : If Key is in List
 -- Found is True.
 -- Location designates the element with Key.
 -- Pred_Loc designates the predecessor of the node designated
 -- by Location. If Location designates the first
 -- node in the list, Pred_Loc is null.
 --
 -- else
 -- Found is False.
 -- Location designates the node that would follow a node
 -- containing Key. If Key's node would follow
 -- the last node in the list, Location is null.
 -- Pred_Loc designates the predecessor of the node designated
 -- by Location. If Location designates the first
 -- first node in the list, Pred_Loc is null.

 Current : Node_Ptr; -- Local copy of Location
 Previous : Node_Ptr; -- Local copy of Pred_Loc
 Node : Node_Type; -- Local copy of a list node
begin
 Current := List; -- Start at the beginning of the linked list
 Previous := Nil; -- No predecessor for 1st element in the list
 loop -- Each iteration, one node is checked
 exit when Current = Nil; -- Reached the end
 Read_Node (Location => Current, Node => Node);
 exit when Key < Key_Of (Node.Info) or -- Passed by
 Key = Key_Of (Node.Info); -- Found it
 Previous := Current; -- Move inchworm's tail
 Current := Node.Next; -- Move inchworm's head
 end loop;
 -- Copy results to out parameters
 Location := Current;

```

```

 Pred_Loc := Previous;
 Found := Current /= Nil and then Key = Key_Of (Node.Info);
end Search_Linked_List;

```

```

-- Operations

```

```

procedure Clear is
 To_Recycle : Node_Ptr; -- Designates a node for recycling
 Node : Node_Type;
begin
 loop
 exit when List = Nil; -- Exit when list is empty
 To_Recycle := List;
 -- Unlink the first node
 Read_Node (Location => List, Node => Node);
 List := Node.Next;
 Free_Node (To_Recycle); -- Recycle the node
 end loop;
end Clear;

```

```

procedure Insert (Item : in Element_Type) is

```

```

 Have_Duplicate : Boolean;
 Pred_Loc : Node_Ptr; -- Pointer to predecessor
 Location : Node_Ptr; -- Pointer to successor
 New_Location : Node_Ptr; -- Pointer to available spot
 Node : Node_Type; -- A list node
begin
 Search_Linked_List (Key => Key_Of (Item),
 Found => Have_Duplicate,
 Pred_Loc => Pred_Loc,
 Location => Location);
 if Have_Duplicate then
 raise DUPLICATE_KEY;
 elsif Pred_Loc = Nil then -- Add at beginning
 New_Location := New_Node;
 Node := (Info => Item, Next => List);
 Write_Node (Location => New_Location, Node => Node);
 List := New_Location;
 else -- Add at the middle or end
 New_Location := New_Node;
 Node := (Info => Item, Next => Location);
 Write_Node (Location => New_Location, Node => Node);
 -- Change the predecessor's Next field
 Read_Node (Location => Pred_Loc, Node => Node);
 Node.Next := New_Location;
 Write_Node (Location => Pred_Loc, Node => Node);
 end if;
 Count := Count + 1;
exception
 when Node_IO.USE_ERROR => -- Raised when disk is full
 raise OVERFLOW;
end Insert;

```

```

procedure Delete (Key : in Key_Type) is
```

```

 Found : Boolean;
 Pred_Loc : Node_Ptr;
 Location : Node_Ptr;
 Next : Node_Ptr;
 Node : Node_Type;
begin
 Search_Linked_List (Key => Key,
 Found => Found,
 Pred_Loc => Pred_Loc,
 Location => Location);

 if not Found then
 raise KEY_ERROR;
 elsif Pred_Loc = Nil then
 Read_Node (Location => Location, Node => Node);
 List := Node.Next; -- Delete first element
 else -- Delete middle or last element
 -- Get the location of the successor of the node to delete
 Read_Node (Location => Location, Node => Node);
 Next := Node.Next;
 -- Change the predecessor to point to the successor
 Read_Node (Location => Pred_Loc, Node => Node);
 Node.Next := Next;
 Write_Node (Location => Pred_Loc, Node => Node);
 end if;
 Free_Node (Location); -- Recycle node memory
 Count := Count - 1;
end Delete;
```

```

procedure Modify (Element : in Element_Type) is
```

```

 Found : Boolean;
 Pred_Loc : Node_Ptr;
 Location : Node_Ptr;
 Node : Node_Type;
begin
 Search_Linked_List (Key => Key_Of (Element),
 Found => Found,
 Pred_Loc => Pred_Loc,
 Location => Location);

 if not Found then
 raise KEY_ERROR;
 else
 Read_Node (Location => Location, Node => Node);
 Node.Info := Element;
 Write_Node (Location => Location, Node => Node);
 end if;
end Modify;
```

```

procedure Retrieve (Key : in Key_Type;
 Element : out Element_Type) is

 Found : Boolean;
 Pred_Loc : Node_Ptr;
 Location : Node_Ptr;
 Node : Node_Type;
begin
 Search_Linked_List (Key => Key,
 Found => Found,
 Pred_Loc => Pred_Loc,
 Location => Location);

 if not Found then
 raise KEY_ERROR;
 else
 Read_Node (Location => Location, Node => Node);
 Element := Node.Info;
 end if;
end Retrieve;

function Empty return Boolean is
begin
 return List = Nil;
end Empty;

function Full return Boolean is
 Node : Node_Type;
 New_One : Node_Ptr;
begin
 if Free /= nil then
 return False;
 else -- See if we can write another node to the disk
 New_One := New_Node;
 Write_Node (Location => New_One, Node => Node);
 Free_Node (New_One); -- Recycle the node
 return False;
 end if;
exception
 when Node_IO.USE_ERROR => -- Raised when disk is full
 return True;
end Full;

function Length return Natural is
begin
 return Count;
end Length;

procedure Traverse is
 Element : Element_Type; -- Use local variable to ensure keys are not changed
 Location : Node_Ptr; -- Designates current node

```

```

 Node : Node_Type;
begin
 Location := List;
 loop
 exit when Location = Nil;
 Read_Node (Location => Location, Node => Node);
 Element := Node.Info; -- Make a local copy of the element
 Process (Element); -- Call user's procedure to process it
 -- Make sure the Key wasn't changed
 if Key_Of (Element) /= Key_Of (Node.Info) then
 raise KEY_ERROR;
 else -- Put possibly modified element back into list
 Node.Info := Element;
 Write_Node (Location => Location, Node => Node);
 end if;
 Location := Node.Next; -- Move to next node
 end loop;
end Traverse;

procedure Close is
begin
 Header_IO.Create (File => Header_File,
 Name => Header_File_Name);
 Header_IO.Write (File => Header_File, Item => Count);
 Header_IO.Write (File => Header_File, Item => List);
 Header_IO.Write (File => Header_File, Item => Free);
 Header_IO.Close (Header_File);
 Node_IO.Close (List_File);
end Close;

begin -- Package Initialization

begin -- Exception block
 Node_IO.Open (File => List_File,
 Mode => Node_IO.InOut_File,
 Name => List_File_Name);
 Header_IO.Open (File => Header_File,
 Mode => Header_IO.In_File,
 Name => Header_File_Name);
 Header_IO.Read (File => Header_File, Item => Count);
 Header_IO.Read (File => Header_File, Item => List);
 Header_IO.Read (File => Header_File, Item => Free);
 Header_IO.Close (Header_File);
exception
 when Node_IO.NAME_ERROR =>
 Node_IO.Create (File => List_File,
 Mode => Node_IO.InOut_File,
 Name => List_File_Name);

 Count := 0;
 List := Nil;
 Free := Nil;
end; -- Exception block

end Sorted_List;

```



### *Analysis of the File-Based List*

Except for the `Length` operation, the Big-O of all operations in the file-based list are the same as those in the array-based linked list. However, Big-O says nothing about the actual amount of time the operations takes. It simply describes how that time changes as the list size grows.  $O(N)$  for a list means that the amount of time to carry out an operation is directly proportional to the size of the list. If we double the size, we double the time.

With a large list, the amount of time an  $O(N)$  file-based operation takes may be significant. Most computer users would not tolerate the wait required to search a large file-based linked list. Even we were too impatient to wait for the results of our timing tests when we used the file-based implementation of the sorted list. In Chapter 12 we look at another linked structure, the binary search tree, whose search is  $O(\log_2 N)$ . The techniques introduced in this chapter for storing linked lists on disk can also be used to implement file-based trees.

### **Summary**

Linked lists are *not* synonymous with dynamic storage allocation. As we demonstrated in this chapter, we can also store linked lists in an array or in a direct file.

We continue to use linked representations of data structures in the chapters that follow and typically implement them with dynamic storage allocation. A static implementation (with arrays and files) of these linked structures is also possible, however, and in some cases desirable or necessary.

### **Exercises**

1. What is the Big-O for initializing the free list in the array-based list package (procedure `Initialize_Memory`)? For the procedures `New_Node` and `Free_Node`?
2. Use the linked lists contained in the array pictured in Figure 10-3 to answer the following questions:
  - (a) What elements (give the `Info` of each) are in `List1`?
  - (b) What elements (give the `Info` of each) are in `List2`?
  - (c) What array positions (indexes) are part of the free-space list?
  - (d) What would the array look like after the deletion of Nell from `List1`?
  - (e) What would the array look like after the insertion of Tracy into `List2`? Assume that before the insertion that the array is as pictured in Figure 10-3.
3. An array of records (nodes) is used to contain a doubly linked list, with the `Next` and `Back` fields indicating the index of the linked nodes in each direction.
  - (a) Show how the array would look after it was initialized to an empty state, with all the nodes linked into the free-space list. (Note that the free-space nodes only have to be linked in one direction.)

| Nodes | .Info | .Next | .Back |
|-------|-------|-------|-------|
| (1)   |       |       |       |
| (2)   |       |       |       |
| (3)   |       |       |       |
| (4)   |       |       |       |
| (5)   |       |       |       |
| (6)   |       |       |       |
| (7)   |       |       |       |
| (8)   |       |       |       |
| (9)   |       |       |       |
| (10)  |       |       |       |

List.First

Free

(b) Draw a box-and-arrow picture of a doubly linked list into which the following numbers are inserted into their proper places in the numerically ordered doubly linked list: 17, 4, 25.

(c) Fill in the contents of the array below after the following numbers were inserted into their proper places in the numerically ordered doubly linked list: 17, 4, 25.

| Nodes | .Info | .Next | .Back |
|-------|-------|-------|-------|
| (1)   |       |       |       |
| (2)   |       |       |       |
| (3)   |       |       |       |
| (4)   |       |       |       |
| (5)   |       |       |       |
| (6)   |       |       |       |
| (7)   |       |       |       |
| (8)   |       |       |       |
| (9)   |       |       |       |
| (10)  |       |       |       |

List.First

Free

(d) Show how the array in part (c) would look after 17 was deleted.

| Nodes | .Info | .Next | .Back |
|-------|-------|-------|-------|
| (1)   |       |       |       |
| (2)   |       |       |       |
| (3)   |       |       |       |
| (4)   |       |       |       |
| (5)   |       |       |       |
| (6)   |       |       |       |
| (7)   |       |       |       |
| (8)   |       |       |       |
| (9)   |       |       |       |
| (10)  |       |       |       |

List.First

Free

4. What is an advantage of implementing a stack as an array-based linked list rather than as a sequential list stored in an array?
5. What is the Big-O for initializing the free list in the file-based list package?
6. Body 10-2 contains a package variable `Count` that is used to store the number of elements in the list. How does this value differ from what a call to the function `Node_IO.Size` returns?
7. We implement the file-based linked list as an ADO rather than an ADT.
  - (a) What changes must we make to the types and subprogram parameters in the package declaration to convert this ADO to an ADT?
  - (b) What additional operation(s) must we include in the package declaration to convert this ADO to an ADT?
  - (c) What additional precondition must we add to all other list operations?

### Programming Problems

1. Using a linked list implemented in an array of records to contain the queue elements, rewrite the FIFO queue private declarations and the package body containing the operations specified in Chapter 7.
2. Implement the sorted list class as an array-based linked list with header and trailer nodes. Turn in the following:
  - (a) written test plan
  - (b) listing of your list package declaration
  - (c) listing of your list package body
  - (d) listing of your test program
  - (e) output from your test program
3. Implement the sorted list class as an array-based circular linked list. Turn in the following:
  - (a) written test plan
  - (b) listing of your list package declaration
  - (c) listing of your list package body
  - (d) listing of your test program
  - (e) output from your test program
4. Answer Exercise 7 and then implement the file-based linked list as an ADT rather than an ADO.
5. The file-based linked list package reads the disk more than it needs to. For example, when we modify an element, the element is read from disk in the search procedure and then read again by the modify procedure.
  - (a) Find other examples in the package where elements are read from the disk multiple times.

We can minimize the number of disk accesses by using a cache of nodes. A cache is a *time-ordered* collection of the most recently read nodes used by procedures `Read_Node` and `Write_Node`. This cache is stored in primary memory (RAM).

Here is one way a cache might operate. When `Read_Node` is called, it first checks to see if the desired node is in the cache. If it is in the cache, it returns a copy—no disk access is done. If it is not in the cache, `Read_Node` reads it from the disk, saves a copy in the cache, and returns a copy to the caller. If the cache is full when adding a node, the oldest node is deleted to make room for the newest node. In our caching scheme, `Write_Node` always writes to the disk and adds a copy of the node to the cache. If that node is already in the cache, the old copy should be deleted and the new copy added.

- (b) What sort of data structure is most appropriate for a cache?
- (c) Write a generic package declaration for a cache ADT that includes all the necessary operations we need to implement our descriptions of the cache-based `Read_Node` and `Write_Node` procedures. Think carefully about what operations to include. You should allow the user to specify a maximum cache size (as a discriminant on the cache type). The cache should be implemented as a linked list.
- (d) Rewrite the package body for the file-based sorted list. Instantiate a cache from your generic cache package. Rewrite procedures `Read_Node` and `Write_Node` to take advantage of the cache.
- (e) Write the package body for your cache ADT.
- (f) Write a test plan and program to test your cache ADT.
- (g) Write a test plan and program to test your file-based list with caching.

You may find that adding caching to the file-based sorted list does not improve its performance. Our Ada programs do not access the computer's disk directly. All input and output is actually performed by the computer's operating system. Our Ada programs do input and output by making requests of the operating system. Because many computer operating systems maintain their own disk caches, adding our own cache may actually slow down the read operations.

## 11— Programming with Recursion

### GOALS

After reading this chapter, you should be able to

- given a recursive routine; determine:  
whether the routine halts the base case(s)  
the general case(s)  
what the routine does whether the routine is correct and, if it is not, correct it
- given a simple recursive problem:  
determine the base case(s)  
determine the general case(s)  
design and code the solution as a recursive procedure or function
- verify a recursive routine, according to the Three-Question Method
- decide whether a recursive solution is appropriate for a problem
- compare and contrast dynamic storage allocation and static storage allocation in relation to using recursion
- explain how recursion works internally by showing the contents of the run-time stack
- replace a recursive solution with iteration and/or the use of a stack
- explain why recursion may or may not be a good choice to implement the solution of a problem

## "Don't Ever Do This!"

As a beginning programmer, you may have been told never to use a function's name within the function on the right-hand side of an assignment statement, as in the following code segment:

```
function Sum return Integer is
 -- Sums five integers from standard input.
 Number : Integer; -- Number read from input
 Result : Integer; -- The sum
begin -- Sum
 Result := 0;
 for Count in 1..5 loop
 Integer_IO.Get (Number);
 Result := Sum + Number
 end loop;
 return Result;
end Sum;
```

You were probably told that using a function name as in the statement `Result := Sum + Number;` would cause something mysterious and undesirable to occur—the function would call itself *recursively*. If we tried to execute

```
Result := Sum + Number,
```

the `Sum` on the *right*-hand side of the assignment would cause function `Sum` to start again through a recursive call. In this case the recursive call is inadvertent; we should have calculated the sum with `Result := Result + Number;`. But there are many situations in which recursion is used on purpose, as a programming tool.

Recursion is a very powerful programming technique that can be used in place of iteration (loops). It involves a different way of looking at repetitive actions—allowing a subprogram to call itself to solve a "smaller" version of its original problem. In this chapter we explore how to understand and write recursive functions and procedures. We also look at how recursion works in a high-level language such as Ada, as well as the factors to consider in choosing an iterative or a recursive solution to a problem.

## The Classic Example of Recursion

Mathematicians often define concepts in terms of the process used to generate them. For instance,  $n!$  (read " $n$  factorial") is used to calculate the number of permutations of  $n$  elements. One mathematical description of  $n!$  is

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n * (n - 1) * (n - 2) * \dots * 1, & \text{if } n > 0 \end{cases}$$

Consider the case of  $4!$ . Because  $n > 0$ , we use the second part of the definition:

$$4! = 4 * 3 * 2 * 1 = 24$$

This description of  $n!$  provides a different definition for each value of  $n$  because the three dots stand in for the intermediate factors. That is, the definition of  $2!$  is  $2 * 1$ , the definition of  $3!$  is  $3 * 2 * 1$ , and so forth.

We can also express  $n!$  with a single definition for any nonnegative value of  $n$ :

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n * (n - 1)!, & \text{if } n > 0 \end{cases}$$

This definition is *recursive* because we express the factorial function in terms of itself.

Let's consider the recursive calculation of  $4!$  intuitively. Because 4 is not equal to 0, we use the second half of the definition:

$$4! = 4 * (4 - 1)! = 4 * 3!$$

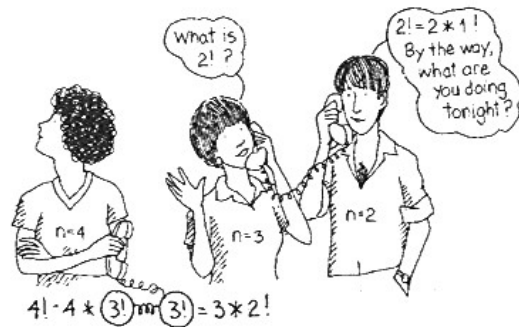
Of course, we can't do the multiplication yet because we don't know the value of  $3!$ . So we call up our good friend Sue Ann, who has a Ph.D. in math, to find the value of  $3!$ .



Sue Ann has the same formula we have for calculating the factorial function, so she knows that

$$3! = 3 * (3 - 1)! = 3 * 2!$$

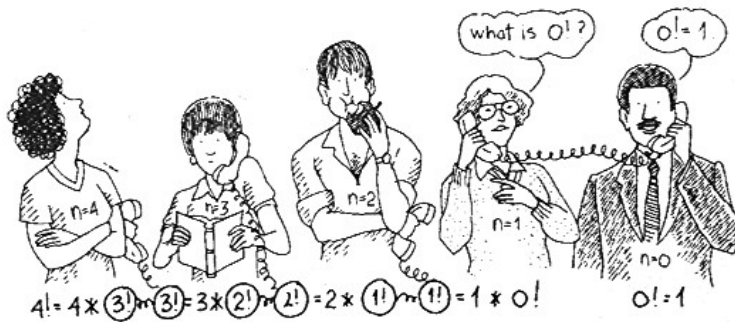
She doesn't know the value of  $2!$ , however, so she puts you on hold and calls up her friend Max, who has an M.S. in math.



Max has the same formula Sue Ann has, so he quickly calculates that

$$2! = 2 * (2 - 1)! = 2 * 1!$$

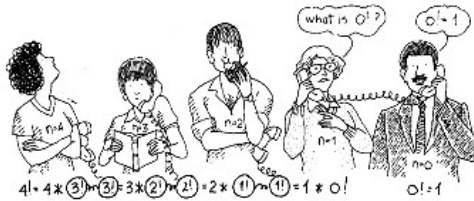
But Max can't complete the multiplication because he doesn't know the value of  $1!$ . He puts Sue Ann on hold and calls up his mother, who has a B.A. in math education.



Max's mother has the same formula Max has, so she quickly figures out that

$$1! = 1 * (1 - 1)! = 1 * 0!$$

Of course, she can't perform the multiplication, because she doesn't have the value of  $0!$ . So Mom puts Max on hold and calls up her colleague Bernie, who has a B.A. in English literature.



Bernie doesn't need to know any math to figure out that  $0! = 1$  because he can read that information in the first clause of the formula ( $n! = 1$ , if  $n = 0$ ). He immediately reports the answer to Max's mother. She can now complete her calculations:

$$1! = 1 * 0! = 1 * 1 = 1.$$

She reports back to Max, who now performs the multiplication in his formula and learns that

$$2! = 2 * 1! = 2 * 1 = 2.$$

He reports back to Sue Ann, who can now finish her calculation:

$$3! = 3 * 2! = 3 * 2 = 6.$$



Sue Ann calls you with this exciting bit of information. You can now complete your calculation:

$$4! = 4 * 3! = 4 * 6 = 24.$$



## Programming Recursively

Of course, the use of recursion is not limited to mathematicians with telephones. Computer languages such as Ada that support recursion give the programmer a powerful tool for solving certain kinds of problems by reducing the complexity or hiding the details of the problem.

We consider recursive solutions to several simple problems. In our initial discussion, you may wonder why a recursive solution would ever be preferred to an iterative, or nonrecursive one, for the iterative solution may seem simpler and more efficient. Don't worry. There are, as you will see later, situations in which the use of recursion produces a much simpler—and more elegant—program.

### Coding the Factorial Function

---

#### Recursive Subprogram

A function or procedure that calls itself.

---

A **recursive subprogram** is one that calls itself. In the previous section Sue Ann, Max, Max's mom, and Bernie all had the same formula for solving the factorial function. When we construct a recursive Ada function `Factorial` for solving  $n!$ , we know where we can get the value of  $(n - 1)!$  that we need in the formula. We already have a function for doing this calculation: `Factorial`. Of course, the actual parameter  $(N - 1)$  in the recursive call is different than the parameter in the original call  $(N)$ . (The recursive call is the one within the function.) As we will see, this different value of the parameter is an important and necessary consideration.

An Ada function for calculating  $n!$  for  $N$  may be coded as follows. Subtype `Natural` is used because the factorial function is only defined for values greater than or equal to zero.

```

function Factorial (N : in Natural) return Positive is
 Result : Positive;
begin
 if N = 0 then -- line 1
 Result := 1; -- line 2
 else
 Result := N * Factorial (N-1); -- line 3
 end if;
 return Result; -- line 4
end Factorial;

```

Notice the use of `Factorial` in line 3. `Factorial` is a recursive call to the function, with the parameter  $N - 1$ .

Let's walk through the calculation of  $4!$  using function `Factorial`. The original value of  $N$  is 4. The steps in the calculation are shown in Table 11-1. These steps are shown graphically in Figure 11-1.

**Table 11-1** Walkthrough of Factorial (4)

| Line | Action                                                                                                                      |
|------|-----------------------------------------------------------------------------------------------------------------------------|
| 1    | $4 \neq 0$ so skip to else clause.                                                                                          |
| 3    | Result := $4 * \text{Factorial}(4 - 1)$<br>First recursive call returns us to the beginning of the function with $N = 3$ .  |
| 1    | $3 \neq 0$ so skip to else clause.                                                                                          |
| 3    | Result := $3 * \text{Factorial}(3 - 1)$<br>Second recursive call returns us to the beginning of the function with $N = 2$ . |
| 1    | $2 \neq 0$ so skip to else clause.                                                                                          |
| 3    | Result := $2 * \text{Factorial}(2 - 1)$<br>Third recursive call returns us to the beginning of the function with $N = 1$ .  |
| 1    | $1 \neq 0$ so skip to else clause.                                                                                          |
| 3    | Result := $1 * \text{Factorial}(1 - 1)$<br>Fourth recursive call returns us to the beginning of the function with $N = 0$ . |
| 1    | $0 = 0$ so go to line 2.                                                                                                    |
| 2    | Result := 1                                                                                                                 |
| 4    | The value of <code>Factorial(0)</code> is returned to the calling statement, the fourth recursive call.                     |
| 3    | Result := $1 * \text{Factorial}(0) = 1 * 1 = 1$                                                                             |
| 4    | The value of <code>Factorial(1)</code> is returned to the calling statement, the third recursive call.                      |
| 3    | Result := $2 * \text{Factorial}(1) = 2 * 1 = 2$                                                                             |
| 4    | The value of <code>Factorial(2)</code> is returned to the calling statement, the second recursive call.                     |
| 3    | Result := $3 * \text{Factorial}(2) = 3 * 2 = 6$                                                                             |
| 4    | The value of <code>Factorial(3)</code> is returned to the calling statement, the first recursive call.                      |
| 3    | Result := $4 * \text{Factorial}(3) = 4 * 6 = 24$                                                                            |
| 4    | The value of <code>Factorial(4)</code> is returned to the calling statement, the original nonrecursive call.                |

For purposes of comparison, let's look at the recursive and iterative solutions to this problem side by side:

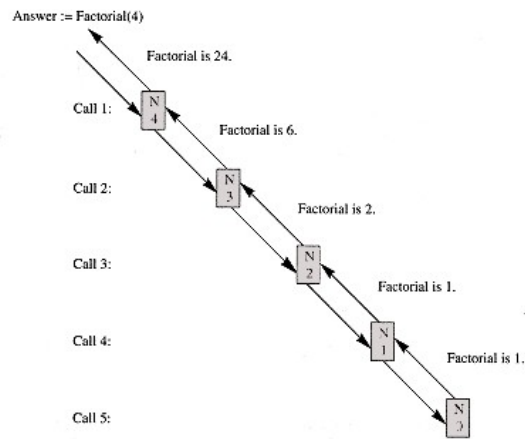


Figure 11-1  
Execution of Factorial (4)

|                                                                                                                                                                                                                                             |                                                                                                                                                                                                                                          |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>function Factorial (N : in Natural)   return Positive is     Result : Positive; begin -- Recursive solution   if N = 0 then     Result := 1;   else     Result := N * Factorial (N-1);   end if;   return Result; end Factorial;</pre> | <pre>function Factorial (N : in Natural)   return Positive is     Result : Positive; begin -- Iterative solution   Result := 1;   for Count in 2..N loop     Result := Result * Count;   end loop;   return Result; end Factorial;</pre> |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

These two versions of `Factorial` illustrate the important difference between recursive and iterative functions. An iterative algorithm uses a *looping construct* such as the `for` loop (or `loop-exit` or `while`) to control the execution. The recursive solution uses a *branching structure* (if-then-else). Recursive subprograms rarely contain loops. In fact, if you use a loop in the design of a recursive algorithm you have probably made an error in your design.

### Verifying Recursive Procedures and Functions

The kind of walkthrough we just did to check the validity of a recursive function or procedure is time consuming, tedious, and often confusing. Furthermore, simulating the execution of `Factorial (4)` tells us that the function works when  $N=4$ , but it doesn't tell us whether the function is valid for *all* values of  $N$ . It would be useful to have a technique that would help us determine inductively whether a recursive algorithm works.

### The Three-Question Method

We use the Three-Question Method of verifying recursive procedures and functions. To verify that a recursive solution works, you must be able to answer yes to all three of these questions:

1. *The Base-Case Question*: Is there a nonrecursive way out of the procedure or function, and does the routine work correctly for this "base" case?
2. *The Smaller-Caller Question*: Does each recursive call to the procedure or function involve a smaller case of the original problem? By smaller, we mean that it is closer to the base case. This guarantees that we ultimately reach the base case.
3. *The General-Case Question*: Assuming that the recursive call(s) works correctly, does the whole procedure or function work correctly?

Let's apply these three questions to function `Factorial`:

1. *The Base-Case Question*: The base case occurs when  $N = 0$ . `Factorial` is then assigned a value of 1, which is the correct value of  $0!$ , and no further (recursive) calls to `Factorial` are made. So the answer is yes.
2. *The Smaller-Caller Question*: To answer this question we must look at the parameters passed in the recursive call. In function `Factorial`, the recursive call passes  $N - 1$ . Each subsequent recursive call sends a decremented value of the parameter, until the value sent is finally 0. At this point, as we verify with the base-case question above, we have reached the base case, and no further recursive calls are made. So the answer is yes.
3. *The General-Case Question*: In the case of a function like `Factorial`, we need to verify that the formula we are using actually results in the correct solution. Assuming that the recursive call `Factorial (N - 1)` gives us the correct value of  $(n - 1)!$ , we get the assignment of  $N * (N - 1)!$  to `Factorial`. This is the definition of a factorial, so we know that the function works for all positive integers. In answering the first question, we have already ascertained that the function works for  $N = 0$ . (The function is defined only for nonnegative integers.) So the answer is yes.

Those of you who are familiar with *inductive proofs* recognize what we have done. Having made the assumption that the function works for some base case  $(n - 1)$ , we can now show that applying the function to the next value,  $(n - 1) + 1$ , or  $n$ , results in the correct formula for calculating  $n!$ .

### Writing Recursive Procedures and Functions

The three questions we use to verify recursive procedures and functions can also be used as a guide for *writing* recursive subprograms. You can use the following approach to write any recursive routine:

- First, get an exact definition of the problem to be solved. (This, of course, is the first step in solving any programming problem.)
- Next, determine the *size* of the problem to be solved on this call to the subprogram. On the initial call to the procedure or function, the size of the whole problem is expressed in the value(s) of the parameter(s).

- Third, identify and solve the *base case(s)* in which the problem can be expressed nonrecursively. This assures a yes answer to the base-case question.
- Last, identify and solve the *general case(s)* correctly in terms of a smaller case of the same problem—a recursive call. This assures yes answers to the smaller-caller and general-case questions.

In the case of `Factorial`, the definition of the problem is summarized in the definition of the factorial function. The size of the problem is the number of values to be multiplied:  $N$ . The base case occurs when  $N = 0$ , in which case we take the nonrecursive path. Finally, the general case occurs when  $N > 0$ , resulting in a recursive call to `Factorial` for a smaller case: `Factorial (N - 1)`.

### Writing a Boolean Function

Let's apply this approach to writing a function, `Value_In_Array`, that searches for a value in an array and returns a Boolean value indicating whether the value has been found. The array is declared as

```
type Array_Type is array (Integer range <>) of Integer;
```

```
My_Array : Array_Type (1..100);
```

The recursive solution to this problem is as follows:

#### Value in Array

```
Result := (Value is in the first position of the array) or
 (Value is in the rest of the array)
return Result
```

We can answer the first question (from our Three-Question Method) just by comparing `Value` to `List (List'First)`. But how do we know whether `Value` is in the rest of the array? If only we had a function that would search the rest of the array. *But we do have one!* Function `Value_In_Array` searches for a value in an array. We simply need to start searching `List` at the second position, instead of the first (a smaller case). To do this, we need to pass the "rest of the array" to `Value_In_Array` as a parameter. We use the following procedure specification:

```
function Value_In_Array (List : in Array_Type,
 Value : in Integer) return Boolean;
-- Searches List for Value
```

```
-- Preconditions : None
```

```
-- Postconditions : returns (Value exists in List)
```

To search the whole array, our program could invoke function `Value_In_Array` with a statement like this one:

```
if Value_In_Array (List => My_Array, Value => My_Value) then
.
.
.
```

The general case of our algorithm is the part that searches the rest of the array. This case involves a recursive call to `Value_In_Array`, specifying a smaller slice of the array to be searched:

```
Result := Value_In_Array (List => List(List'First+1 .. List'Last),
 Value => Value);
```

By passing a slice beginning with the second element of the array, we have diminished the size of the problem to be solved by the recursive call. That is, searching the array from `List'First + 1` to `List'Last` is a smaller task (smaller by one array element) than searching from `List'First` to `List'Last`.

Finally, we need to know when to stop searching. In this problem we have two base cases: (1) when the value is found (return `True`), and (2) when the array contains no elements (return `False`). In either case we can stop making recursive invocations to `Value_In_Array`.

Let's summarize what we have discussed and then write function `Value_In_Array`.

### Function `Value_In_Array`

**Definition:** Search `List` for `Value`. Return `True` if `Value` is found; return `False` otherwise.

**Size:** The number of elements in `List` (`List'Length`).

**Base Cases:** (1) If `List(List'First) = Value`, return `True`.

(2) If `List'Length = 0`, return `False`.

**General Case:** Search the rest of the list for `Value`. This is a recursive invocation of `Value_In_Array` with a slice (smaller caller).

The code for function `Value_In_Array` is given below. Figure 11-2 shows an execution of this function.

```
function Value_In_Array (List : in Array_Type,
 Value : in Integer) return Boolean is
 Result : Boolean;
begin
 if List'Length = 0 then
 Result := False; -- Base Case 2, empty array
 elsif Value = List(List'First) then
 Result := True; -- Base Case 1, value found
 else -- General Case
 Result := Value_In_Array -- Search rest of List
 (List => List (List'First+1 .. List'Last),
 Value => Value);
 end if;
 return Result;
end Value_In_Array;
```

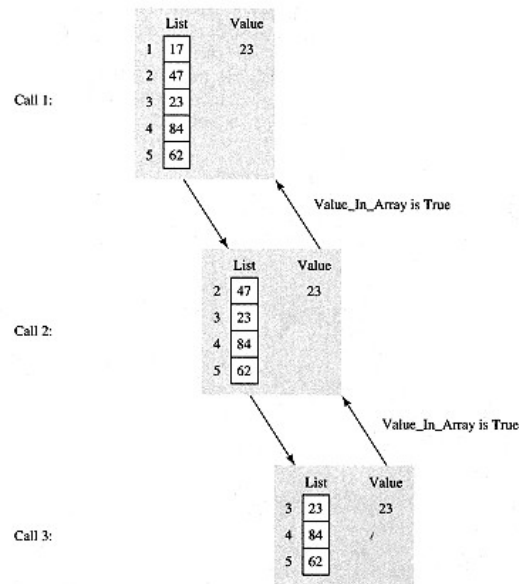


Figure 11-2  
Execution of Value\_In\_Array

The equivalent iterative solution would use a local counter, initialized inside the function before the loop and incremented by an assignment statement inside the loop.

### Multiple Recursive Calls

All of our previous examples of recursion contain a single recursive call. Recursive algorithms can contain several different base cases and several different recursive calls. Let's look at such an algorithm. The following sequence of numbers is called the Fibonacci sequence:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, . . .

The numbers in this sequence are called Fibonacci numbers. Each Fibonacci number in this sequence is the sum of the two previous numbers. Fibonacci numbers are often found in nature. In fact, the sequence was first used by Leonardo Fibonacci of Pisa, a medieval mathematician, to determine the number of rabbits that would be produced in a hutch in  $N$  months starting with a single pair of newborn rabbits. Leonardo made the following assumptions in this problem:

1. Newborn rabbits become adults (are able to breed) one month after birth.
2. There are an equal number of male and female rabbits.
3. Each pair of adult rabbits produces one pair of newborn rabbits each month.
4. No rabbits die.

Figure 11-3 illustrates the pattern of reproduction in the hutch for seven months. Each row of the figure traces one pair of rabbits through the months. For example, the first row traces the initial pair of rabbits. The pair is young during the first month and adult in the remaining months. The ninth row traces the first pair of offspring of the initial pair. The number of rows under any given month is the number of pairs in the hutch that month. The monthly numbers are listed at the bottom of the figure.

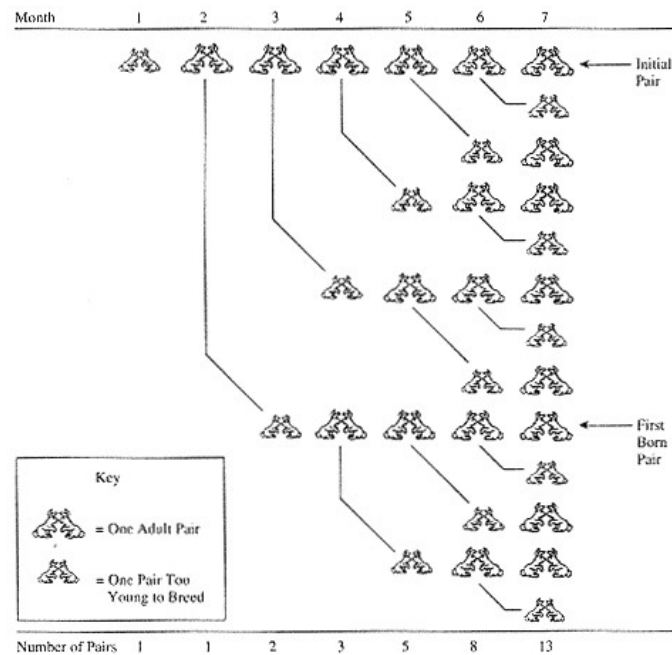


Figure 11-3  
The Pattern of Rabbit Reproduction over Seven Months

The number of pairs of rabbits each month is equal to the number of pairs last month plus the number of pairs born this month (which is equal to the number of pairs two months previous). This is a Fibonacci sequence. To know how many rabbits are in the hutch after a year, we just determine the thirteenth Fibonacci number.

*The Base-Case Question:* The first number in the Fibonacci sequence is always 1. This number looks like a good base case. We can state it as

$$F_1 = 1$$

*The General-Case Question:* The value of the  $n$ th Fibonacci number is equal to the sum of the two previous Fibonacci numbers. We can state this relation as



$$F_n = F_{n-1} + F_{n-2}$$

*The Smaller-Caller Question:*  $F_n$  is calculated from two smaller Fibonacci numbers,  $F_{n-1}$  and  $F_{n-2}$ . For example,  $F_5$  is calculated from  $F_4$  and  $F_3$ .

There is a problem in using our general case for calculating  $F_2$ . Calculating  $F_2$  requires the value of the first Fibonacci number and the zeroth Fibonacci number, a number which is not defined. The solution to this problem is to add a *second base case*. Both the first and second Fibonacci numbers are 1. Here is a summary of our two base cases and general case containing two recursive calls:

### Fibonacci

```

if N is 1 or 2 then -- Two base cases
 the result is 1
else
 the result is Fibonacci (N - 1) + Fibonacci (N - 2) -- Two recursive calls

```

And here is an Ada function to determine a particular Fibonacci number.

```

function Fibonacci (N : in Positive) return Positive is
-- Determine the Nth Fibonacci number
 Result : Positive;
begin
 if N <= 2 then -- Two base cases
 Result := 1;
 else
 Result := Fibonacci (N-1) + Fibonacci (N-2); -- Two recursive calls
 end if;
 return Result;
end Fibonacci;

```

### Using Recursion to Simplify Solutions

So far the examples we have looked at could just as easily (or more easily) have been written as iterative routines. At the end of the chapter, we talk more about choosing between iterative and recursive solutions. There are many problems, however, in which using recursion simplifies the solution.

The first problem we look at is a function, `Combinations`, that tells us how many combinations of a certain size can be made out of a total group of elements. For instance, if we have 20 different books to pass out to four students, we can easily see that—to be equitable—we should give each student five books. But how many combinations of five books can be made out of a group of 20 books?

The optimal solution of our language translation case study in Chapter 4 involved combinations. We asked how many combinations of two languages, of three languages, of four languages, and so on can we make out of the group of ten languages spoken.

There is a mathematical formula that can be used for solving this problem. Given that  $C$  is the total number of combinations,  $Group$  is the total size of the group to pick from,  $Members$  is the size of each subgroup, and  $Group \geq Members$ ,

$$\{$$

Because this definition of `Combinations` is recursive, it is easy to see how a recursive function could be used to solve the problem.

Let's summarize our problem.

### Function Combinations

**Definition:** Calculate how many combinations of  $Members$  size can be made from the total  $Group$  size.

**Size:** Size of  $Group$ ,  $Members$ .

**Base Cases:** (1) If  $Members = 1$ ,  $Combinations := Group$

(2) If  $Members = Group$ ,  $Combinations := 1$

**General Case:**  $Combinations := Combinations(Group - 1, Members - 1) + Combinations(Group - 1, Members)$

The resulting recursive function, `Combinations`, is listed below.

```
function Combinations (Group : in Positive;
 Members : in Positive) return Positive is
-- Returns the number of combinations of Members size can be
-- made from the total group size.
 Result : Positive;
begin
 if Members = 1 then
 Result := Group; -- Base Case 1
 elsif Members = Group then
 Result := 1; -- Base Case 2
 else
 Result := Combinations (Group - 1, Members - 1) +
 Combinations (Group - 1, Members);
 end if;
 return Result;
end Combinations;
```

The processing of this function to calculate the number of combinations of three elements that can be made from a set of four is shown in Figure 11-4.

Returning to our original problem, we can now find out how many combinations of five books can be made from the original set of 20 books with the statements

```
Text_IO.Put ("Number of combinations ");
Integer_IO.Put (Item => Combinations (Group => 20, Members => 5));
```

Writing a recursive solution to a problem that is characterized by a recursive definition, like `Combinations` or `Factorial` above, is fairly straightforward. In the next sections we look at some problems that have no recursive definitions; we must design our own recursive solution as we did for functions `Value_In_Array` and `Fibonacci`.

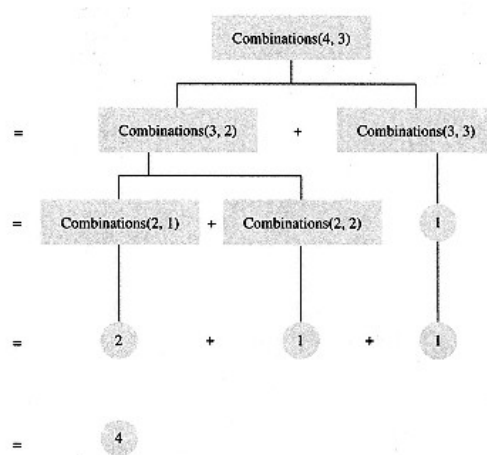


Figure 11-4  
Calculating Combinations (4, 3)

### Recursive Processing of Linked Lists

Let's look at a procedure that traverses a linked list. The linked list has been implemented using the following declarations:

```
type Node_Type;
```

```
type Node_Ptr is access Node_Type;
subtype List_Type is Node_Ptr;
```

```
type Node_Type is
 record
 Info : Integer;
 Next : Node_Ptr;
 end record;
```

```
My_List: List_Type; -- External pointer to first node of list
```

By now you are probably protesting that this task is so simple to accomplish iteratively (loop, exit when `Ptr = null`) that it does not make any sense to write it recursively. So let's make the task more fun: Process the elements in the list in *reverse* order. This problem is much more easily and elegantly solved recursively.

What is the task to be performed? The algorithm is given below and illustrated in Figure 11-5.

### Reverse Traverse

Traverse the second through last elements in the list in reverse order  
Process the first element in the list

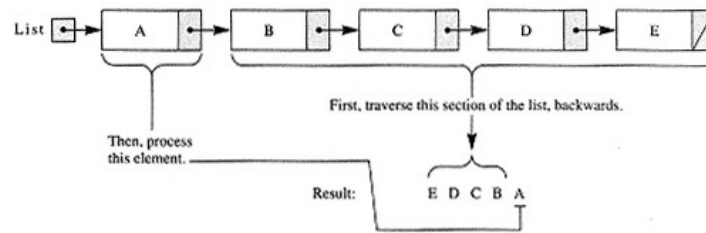


Figure 11-5  
Recursive Reverse\_Traverse

The second part of the task is simple. Because `List` points to the first node in the list, we can process it with the statement `Process (List.all.Info);`. The first part of the task—traversing all the other nodes in the list in reverse order—is also simple because we have a routine that traverses lists in reverse order: We just call procedure `Reverse_Traverse` recursively. Of course, we have to adjust the parameter somewhat, to `Reverse_Print (List => List.all.Next)`. This call says, "Traverse, in reverse order, the linked list pointed to by `List.all.Next`." This task in turn is accomplished recursively in two steps:

Reverse\_Traverse the rest of the list (third through last elements).  
Then Process the second element in the list.

And of course the first part of this task is accomplished recursively. Where does it all end? We need a base case. What is the simplest list to traverse? A list with only one element is easy to traverse, but a list with no elements is even easier. The base case for most all recursive linked-list algorithms is an empty list. Let's summarize the problem.

### Procedure Reverse\_Traverse

**Definition:** Traverse the list in reverse order.

**Size:** Number of elements in the list pointed to by `List`.

**Base Case:** If the list is empty, do nothing.

**General Case:** Reverse\_Traverse the list pointed to by `List.all.Next`, then process `List.all.Info`.

The other recursive routines that we have written have been functions; the `Reverse_Traverse` operation is a procedure.

```

procedure Reverse_Traverse (List : in List_Type) is
-- Traverses the list in reverse order.
-- This is a recursive solution.

```

```

begin
 if List /= null then
 Reverse_Traverse (List => List.all.Next); -- General
 Process (Element => List.all.Info); -- case
 end if;
 -- else list is empty (do nothing) -- Base case
end Reverse_Traverse;

```

To traverse (in reverse order) the whole linked list designated by `My_List`, we would use the following procedure call:

```
Reverse_Traverse (List => My_List).
```

Let's verify this procedure using the Three-Question Method.

1. *The Base-Case Question:* The base case is implied. When `List = null`, we return to the statement following the last recursive call to `Reverse_Traverse`, and no further recursive calls are made. So the answer is yes.
2. *The Smaller-Caller Question:* The recursive call passes the list pointed to by `List.all.Next`, which is one node smaller than the list pointed to by `List`. So the answer is yes.
3. *The General-Case Question:* We assume that `Reverse_Traverse (List => List.all.Next)` correctly traverses the rest of the list in reverse order; this call, followed by the statement processing the value of the first element, gives us the whole list, traversed in reverse order. So the answer is yes.

Exercise 6 asks you to change procedure `Reverse_Traverse` to make it traverse the list in order.

### A Recursive Version of Binary Search

In the application section of Chapter 2, we developed the binary search procedure. Let's review the algorithm.

#### Binary\_Search

```

if there is nothing remaining in the list to examine
 stop searching - value not in list
else
 Examine the middle element in the array
 if the middle element contains the desired key then
 stop searching - value in list
 elsif the middle element is larger than the desired key then
 Search the first half of the array
 else (middle element is smaller than the key)
 Search the second half of the array
 end if
end if

```

Though the procedure that we wrote in Chapter 2 was iterative, our *thinking* during its design was recursive. The solution is expressed in smaller versions of the original problem:

If the answer isn't found in the middle position, search the appropriate half of the array (a smaller problem). Let's summarize the problem using our recursive terminology.

### Procedure Binary\_Search

**Definition:** Search the array for position of Value.

**Size:** The number of elements in the array.

**Base Cases:** (1) If List'Length = 0, Location := 0.  
(Note: List'Length is 0 when List'First > List'Last)  
  
(2) If List(Midpoint) = Value, Location := Midpoint.

**General Cases:** (1) If List(Midpoint) > Value, Binary\_Search the first half of List.  
  
(2) If List(Midpoint) < Value, Binary\_Search the second half of List.

This problem has two different base cases (we find Value or we don't find Value) and two general cases (we search the left half or the right half of the list). Here is the recursive version of the procedure. Figure 11-6 shows its execution with an array containing five integers.

```
procedure Binary_Search (List : in Array_Type;
 Value : in Element_Type;
 Location : out Natural) is
-- Search List for Value;
-- return Location (an array index) if found,
-- otherwise return value of 0 in Location.

 Midpoint : Natural; -- Index of search area's midpoint

begin -- Binary_Search
 if List'Length = 0 then
 Location := 0; -- Base Case 1
 else
 Midpoint := (List'First + List'Last) / 2;
 if List (Midpoint) = Value then
 Location := Midpoint; -- Base Case 2
 elsif List (Midpoint) > Value then
 -- Search first half of the list (General Case 1)
 Binary_Search (List => List (List'First..Midpoint-1),
 Value => Value,
 Location => Location);
 else
 -- Search second half of the list (General Case 2)
 Binary_Search (List => List (Midpoint+1..List'Last),
 Value => Value,
 Location => Location);
 end if;
 end if;
end Binary_Search;
```

How does this version of the code compare with the iterative version in Chapter 2? We'll look at the objective measures first: In terms of non-comment lines of code, the

recursive version is nearly 20 percent shorter. The iterative version used four local variables, whereas the recursive version uses only one. Score one point for the recursive version. Timed execution experiments (detailed later in this chapter) show that for one particular computer, the iterative binary search procedure runs about twice as fast as the recursive binary search procedure. Score one point for the iterative version.

The subjective measure asks, which procedure is easier to write and to understand? We think that the recursive procedure wins this round. It works the way that we naturally think of the problem: divide and conquer. Therefore our final score is recursive version two points, iterative version one point.

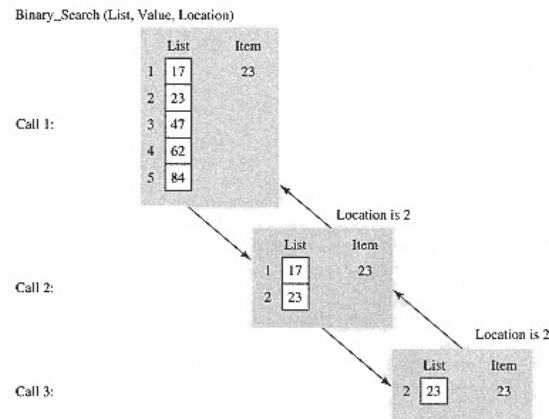


Figure 11-6  
Execution of Binary\_Search

## How Recursion Works

### Binding

The association of a memory address with a variable name.

### Binding Time

The point in the compile/link/execution cycle when variable names are associated with addresses in memory.

In order to understand how recursion works and why some programming languages allow it and some do not, we have to take a detour and look at how languages associate places in memory with variable names. The association of a memory address with a variable name is called **binding**. The point in the compile/link/execute cycle when binding occurs is called the **binding time**. Appendix J describes the compile/link/execution cycle. We want to stress that binding time refers to a point of time in a process, not the amount of clock time that it takes to bind a variable.

Programming languages are usually classified as having either static storage allocation or dynamic storage allocation. Static storage allocation associates variables with memory locations at compile or link time; dynamic storage allocation associates variables with memory locations at execution time. We know from our discussions of elaboration and access variables that Ada is a language that supports dynamic storage allocation. As we look at how static and dynamic storage allocation work, consider the following question: *When are the parameters of a procedure bound to a particular address in memory?* The answer to this question tells something about whether recursion can be supported.

### Static Storage Allocation

As a program is being translated, the compiler creates a table called a *symbol table*. When a variable is declared, it is entered into the symbol table, and a memory location—an address—is assigned to it. For example, let's see how the compiler would translate the following Ada declarations:

```
Girls : Natural;
Boys : Natural;
Kids : Natural;
```

To simplify this discussion, we assume that integers take only one memory location. These three declarations cause three entries to be made in the symbol table:

| <i>Symbol</i> | <i>Address</i> |
|---------------|----------------|
| Girls         | 0000           |
| Boys          | 0001           |
| Kids          | 0002           |

That is, *at compile time*,

Girls is *bound* to Address 0000.

Boys is *bound* to Address 0001.

Kids is *bound* to Address 0002.

Whenever a variable is used later in the program, the compiler searches the symbol table for its actual address, and substitutes that address for the variable name. After all, meaningful variable names are for the convenience of the human reader; addresses, however, are meaningful to computers. For example, the assignment statement

```
Kids := Girls + Boys;
```

is translated into machine instructions that execute the following actions:

- Get the contents of Address 0000.
- Add it to the contents of Address 0001.
- Put the result into Address 0002.

The object code (machine language instructions translated from the source program by the compiler) are then stored in a different part of memory. Let's say that the machine instructions begin at Address 1000. At the beginning of the program, control is transferred to Address 1000. The instruction stored there is executed; then the instruction in 1001 is executed; and so on.

Where are the parameters of subprograms stored? With static storage allocation, the formal parameters of a subprogram are assumed to be in a particular place; for instance, the compiler might set aside space for the parameter values immediately preceding the code for each subprogram. The procedure below



```

procedure Count_Kids (Girl_Count : in Natural;
 Boy_Count : in Natural) is
 Total_Kids : Natural;
.
.
.

```

has two parameters, `Girl_Count` and `Boy_Count`, as well as a local variable `Total_Kids`. Let's assume that the procedure's code begins at an address we call `Count_Kids`. The compiler leaves room for the two formal parameters and the local variable at addresses `Count_Kids - 1`, `Count_Kids - 2`, and `Count_Kids - 3`, respectively. The statement

```

Total_Kids := Girl_Count - Boy_Count;

```

in the body of the procedure would generate the following actions:

- Get the contents of Address `Count_Kids - 1`.
- Add it to the contents of Address `Count_Kids - 2`.
- Store the result in Address `Count_Kids - 3`.

Figure 11-7 shows how a program with two subprograms might be arranged in memory when static storage allocation is used. The machine instructions for the main program, procedure `Print_Total`, and procedure `Count_Kids` begin at locations 1000, 4400, and 9800, respectively. The formal parameters `Girl_Count` and `Boy_Count` are at addresses 9799 (`Count_Kids - 1`) and 9798 (`Count_Kids - 2`). The local variable `Total_Kids` is at address 9797 (`Count_Kids - 3`).

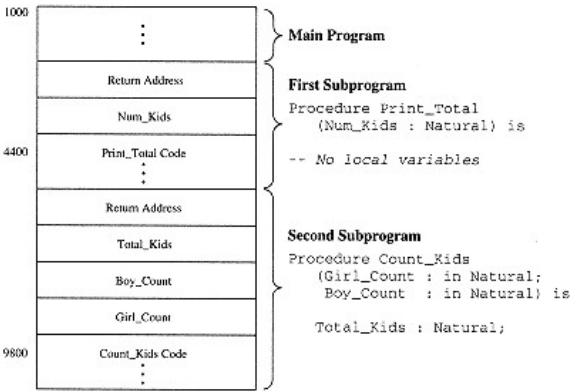


Figure 11-7  
Static Allocation of Space for a Program with Two Subprograms

This discussion has been greatly simplified, since among other things, the compiler sets aside space not only for the parameters and local variables, but also for the return address (the location in memory of the next instruction to process, following the completion of the procedure) and the computer's current register values. However, we have illustrated the main point. The addresses of the procedure's formal parameters and local variables are fixed *before the program runs*.

We can compare the static allocation scheme to one way of allocating seats in an auditorium where a lecture is to be held. A finite number of invitations are issued for the event, and the exact number of chairs needed are set up before the lecture. Each invited guest has a reserved seat. So if anyone brings friends, there is nowhere for them to sit.

What is the implication of binding variables to memory locations before the program executes? Each parameter and local variable has but a single location assigned to it at compile time. (They are like invited guests with reserved seats.) If each call to a procedure is an independent event, there is no problem. But in the case of recursion, each recursive call is dependent on the state of the values in the previous call. Where is the storage for the multiple versions of the parameters and local variables generated by recursive calls? Because the intermediate values of the parameters and local variables must be retained, the recursive call cannot store its arguments in the fixed number of locations that were set up at compile time. The values from the previous recursive call would be overwritten and lost. Thus a language that uses only static storage allocation *cannot* support recursion.

### **Dynamic Storage Allocation**

The situation described above is like a class of students who must share one copy of a workbook. Joe writes his exercise answers in the space provided in the workbook, then Mary erases his answers, and writes hers in the same space. This process continues until each student in the class writes his or her answers into the workbook, obliterating all the answers that came before. Obviously this situation isn't practical. Clearly what is needed is for each student to read from the single copy of the workbook, then to write his or her answers on a separate piece of paper. In computer terms, what each invocation of a procedure needs is its own workspace. Dynamic storage allocation provides this solution.

---

#### **Relative Address**

The offset (number of memory locations) from some other address determined at run time.

---

With dynamic storage allocation, variables are not bound to actual addresses in memory until *run time*. The compiler references variables not by their actual addresses, but by **relative addresses**, the offset from some other address known only at run time. Of particular interest to us, the compiler references the parameters and local variables of a procedure or function relative to some address known only at run time, not relative to the location of the subprogram's code.

Let's look at a simplified version of how this might work in Ada. (The actual implementation depends on the particular machine and compiler.) When a subprogram is invoked, it needs space to keep its formal parameters, its local variables, and the return address (the address in the calling program to which the computer returns when the subprogram completes its execution). Just like students sharing one copy of a workbook, each invocation of a procedure or function needs its own workspace. This workspace is called an *activation record*. A simplified version of an activation record for function `Factorial` might have the following "declarations":

```

type Activation_Rec is
 record
 Return : Address_Type; -- return address
 Factorial : Natural; -- return value
 N : Natural; -- formal parameter
 Result : Natural; -- local variable
 end record;

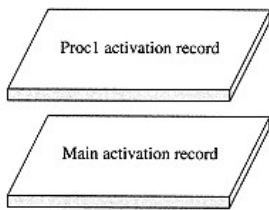
```

Each call to a procedure or function, including recursive calls, generates a new activation record. Within the subprogram, references to the parameters and local variables use the values in the activation record. When the subprogram ends, the activation record is released. How does this happen? Your source code doesn't need to allocate and free activation records; the compiler adds a "prologue" to the beginning of each procedure and an "epilogue" to the end of each procedure. Execution of the prologue is what Ada calls elaboration. The next table compares the source code for function `Factorial` with a simplified version of the "code" executed at run time. (Of course the code executed at run time is in machine language, but we are listing the source code "equivalent" so that it makes sense to the reader.)

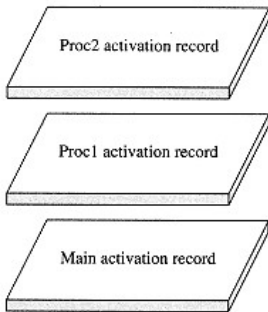
#### Run-Time Version of Factorial (Simplified)

| <i>What Your Source Code Says</i>                                                                                                                  | <i>What the Run-Time System Does</i>                                                                                                                                                                                                                                                              |
|----------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> function Factorial   (N : in Natural)   return Positive is     Result : Positive; </pre>                                                     | <pre> -- Function Prologue (elaboration) Act := new Activation_Rec; Act.all.Return := Ret_Address; Act.all.N := N; </pre>                                                                                                                                                                         |
| <pre> begin   if N = 0 then     Result := 1;   else     Result :=       N *       Factorial(N-1);   end if;   return Result; end Factorial; </pre> | <pre> if Act.all.N = 0 then   Act.all.Result := 1; else   Act.all.Result :=     Act.all.N *     Factorial (Act.all.N-1); end if Act.all.Factorial := Act.all.Result -- Function Epilogue Factorial.all := Act.all.Factorial; Ret_Addr := Act.all.Return; Free (Act); Jump (goto) Ret_Addr; </pre> |

What happens to the activation record of one subprogram when a second subprogram is invoked? Consider a program whose main program calls `Proc1`, which then calls `Proc2`. When the program begins executing, the "main" activation record is generated. (Because the main program's activation record exists for the entire execution of the program, the program's global data can be considered "static.") At the first procedure call an activation record is generated for `Proc1`:



When Proc2 is called from within Proc1, its activation record is generated. Since Proc1 has not finished executing, its activation record is still around; just like the mathematicians with telephones, one waits "on hold" until the next call is finished:



When Proc2 finishes executing, its activation record is released. But which of the other two activation records becomes the active one: Proc1's or Main's? Proc1's activation record is now active, of course. The order of activation follows the Last-In-First-Out rule. We know of a structure that supports LIFO access—the stack—so it should come as no surprise that the structure that keeps track of the activation records at run time is called the *run-time stack*.

When a procedure or function is invoked, its activation record is pushed onto the runtime stack. Each nested level of procedure calls adds another activation record to the stack. As each subprogram completes its execution, its activation record is popped from the stack. Recursive procedure or function calls, like calls to any other subprograms, cause a new activation record to be generated. The level of recursive calls in a program determines how many activation records for this subprogram are pushed onto the run-time stack at any one time.

Using dynamic allocation might be compared to another way of allocating seats in an auditorium where a lecture has been scheduled. A finite number of invitations is issued, but each guest is asked to bring his or her own chair. In addition, each guest can invite an unlimited number of friends, as long as they all bring their own chairs. Of course, if the number of extra guests gets out of hand, the space in the auditorium runs out, and there is

not enough room for any more friends or chairs. Similarly, the level of recursion in a program must eventually be limited by the amount of memory available in the run-time stack.

Let's walk through function `Factorial` again, to see how its execution affects the run-time stack. Here is the function:

```
function Factorial (N : in Natural) return Positive is
 Result : Positive;
begin
 if N = 0 then
 Result := 1;
 else
 Result := N * Factorial (N-1);
 end if;
 return Result;
end Factorial;
```

Let's say that the main program is loaded in memory beginning at location 5000, and that the initial call to `Factorial` is made in a statement at memory location 5200. The `Factorial` procedure is loaded in memory at location 1000, with the recursive call made in the statement at location 1010. Figure 11-8 shows a simplified version of how this example program is loaded in memory. (These numbers have been picked arbitrarily so that we have actual numbers to show in the Return address field of the activation record.)

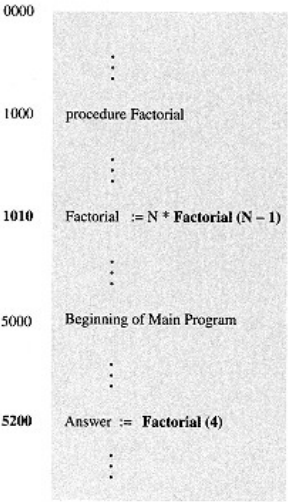
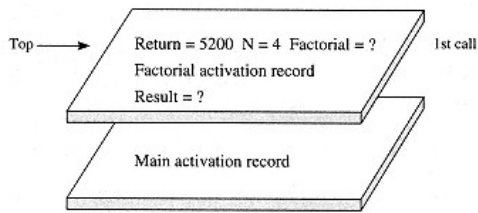


Figure 11-8  
The Sample Program Loaded in Memory

When `Factorial` is called the first time from the statement in the main program at address 5200,

```
Answer := Factorial (4);
```

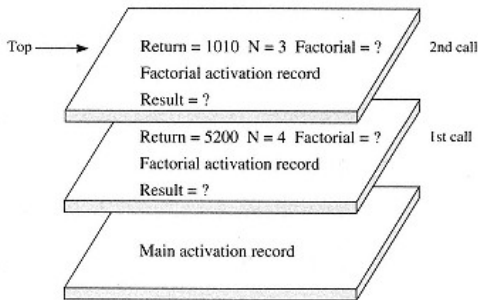
an activation record is pushed onto the run-time stack to hold four pieces of data: the return address (5200), the value that is returned from the function (Factorial), the formal parameter N (4), and the local variable Result. This activation record is now on the top of the run-time stack:



The code is now executed. Is N (the N value in the top activation record) equal to 0? No, it is 4, so the else branch is taken:

```
Factorial := N * Factorial(N - 1);
```

This time the function Factorial is called from a different place. It is called recursively from within the function, from the statement at location 1010. After the value of Factorial(N - 1) is calculated, we return to this location to calculate the value returned from the function. A new activation record is pushed onto the run-time stack:



Again the code is executed. Is N (the value in the top activation record) equal to 0? No, N = 3, so the else branch is taken:

```
Factorial := N * Factorial(N - 1);
```

So the function `Factorial` is again called recursively from the instruction at location 1010. This process continues until the situation looks as shown in Figure 11-9. Now, as the code is being executed, we again ask the question: Is `N` (the value of `N` in the top activation record) equal to 0? *Yes*. This time we take the `then` branch, storing the value 1 in `Result` (the `Result` in the top activation record, that is). The statement

```
return Result;
```

assigns the value of `Result` to `Factorial`. The fifth invocation of the function has executed to completion, and the value of `Factorial` in the top activation record is returned from the function. The run-time stack is popped to release the top activation record, leaving the activation record of the fourth call to `Factorial` at the top of the run-time stack. We don't restart the function from the beginning, however. As with any procedure or function call, we return to execute the instruction following the call to `Factorial`. This was the return address (location 1010) stored in the activation record.

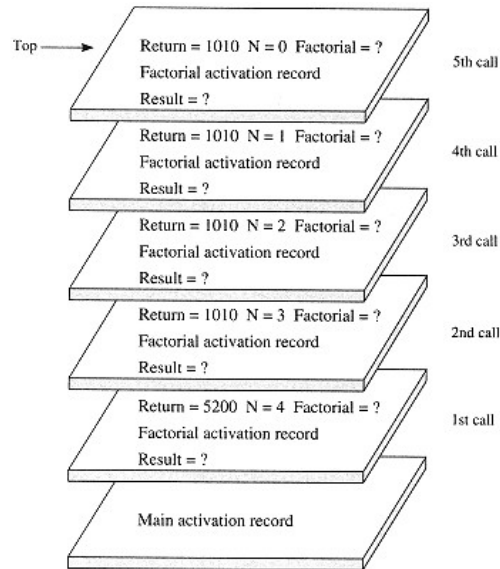
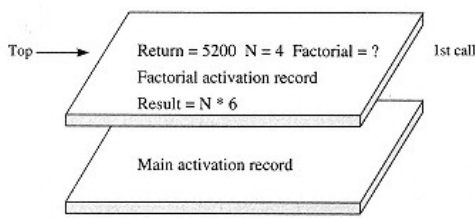


Figure 11-9  
The Run-Time Stack During the Execution of `Factorial`

The next instruction is where the returned value (1) is multiplied by the value of `N` in the top activation record (1) and the result (1) is stored in `Result` (the instance of `Result` in the top activation record, that is). The return statement assigns `Result` to `Factorial`. Now the fourth invocation of the function is complete, and the value of `Factorial` in the top activation record is returned from the function. Again the run-time stack is popped to release the top activation record, leaving the activation record of the

third call to `Factorial` at the top of the run-time stack. We return to execute the instruction following the recursive call to `Factorial`.

This process continues until we are back to the first call, and 6 has just been returned as the value of `Factorial (N - 1)`. This value is multiplied by the value of `N` in the top activation record (that is, 4) and the result, 24, is stored in the `Result` field of the top activation record. The assignment of `Result` to `Factorial` by the return statement completes the execution of the initial call to function `Factorial`. The value of `Factorial` in the top activation record (24) is returned to the place of the original call, and the activation record is popped. This leaves the main activation record at the top of the run-time stack. The final value of `Factorial` is stored in the variable `Answer`, and the statement following the original call is executed.



### Parameter Passing

The fact that the parameters for the recursive calls have to be passed in their activation records suggests that we should avoid passing large structures as parameters. But this is not the case. Ada parameter modes are designed to enable the compiler to minimize the amount of activation record storage needed for parameters.

Let's review Ada's three parameter modes. Mode `in` is used to pass information from the caller to the subprogram. Mode `out` is used to return results from the subprogram to the caller. And mode `in out` is used for the times we need a subprogram to modify values supplied by the caller; a value is passed in, changed, and returned. We select parameter modes based on the direction of information flow between the caller and the subprogram.

When you first studied subprogram parameters you may have learned that information is exchanged between actual and formal parameters by copying. This model is valid for scalar value parameters such as integers, floats, characters, and enumeration types. However, when a structured type like an array or record is passed, the entire structure isn't copied. Instead, the subprogram is supplied with the address of the original data. The formal and actual parameters become synonyms for the same memory locations.\* So when the subprogram changes the value of a formal structured parameter, it also changes the actual parameter. But even though the subprogram is manipulating the original data, the restrictions imposed by parameter modes are still enforced; a subprogram cannot modify an `in` mode parameter.

When designing the interface for a subprogram, you shouldn't worry about *how* the information is actually transferred between the caller and the subprogram. Continue to select parameter modes based on the direction of information flow needed by the problem. The Ada compiler ensures that the parameters are passed in the most efficient way possible.

---

\*This parameter-passing mechanism is often called *pass by reference*.



## Debugging Recursive Routines

Because of their nested calls to themselves, recursive routines can be confusing to debug. The most serious problem is the possibility that the routine recurses forever. A `STORAGE_ERROR` exception is the symptom of this problem, telling us that the system has run out of space in the run-time stack due to the level of recursive calls. Using the Three-Question Method to verify recursive procedures and functions should help us avoid the problem of never finishing. If we can answer yes to the base-case and smaller-caller questions, we should be able to guarantee that the routine eventually ends—theoretically, at least.

That does not guarantee, however, that the program does not fail due to lack of space. In the previous section we saw that a procedure call requires a certain amount of overhead to save the parameters, the return address, and the local data. A call to a recursive procedure may generate many, many levels of procedure calls to itself—more than the system can handle.

One error that programmers often make when they first start writing recursive routines is to use a looping structure instead of a branching one. Because they tend to think of the problem in terms of a repetitive action, they inadvertently use a loop rather than an if statement. The main body of the recursive routine should always be a breakdown into base and recursive cases. Hence, we use a branching statement, not a looping statement. It's a good idea to double check your recursive subprograms to make sure that you are using an if or case statement to get a branching effect.

Recursive routines are good places to put debug Puts (or debugger breakpoints) during testing. Display the parameters and local variables, if any, at the beginning and end of the subprogram. Be sure to display the values of the parameters on the recursive call(s) to make sure that each call is trying to solve a problem smaller than the previous one.

The beauty of recursion is that it can easily be verified *before* the code is even entered. The best way to debug recursive routines is to design them so there are no bugs to find. That's what our Three-Question Method is for.

## Removing Recursion

In cases where a recursive solution isn't desirable, either because the language doesn't support recursion or because the recursive solution is deemed too costly in terms of space (multiple activation records) or time (on many computers calling a subprogram takes more time than evaluating a loop termination condition), a recursive algorithm can be implemented as a nonrecursive procedure or function. There are two general techniques that are often substituted for recursion: iteration and stacking.

### Iteration

As we note later in this chapter, function `Value_In_Array` is a poor use of recursion. However, it is simple to remove the recursion from this kind of subprogram—one in which the recursive call is in the last statement executed in the general case. Let's see how to replace this recursion with a loop.

The recursive solution has two base cases: One occurs if we find the value, and the other occurs if we reach the end of the list without finding the value. The base cases solve the problem without further executions of the function. In the iterative solution, the base cases become the terminating conditions of the loop:

```

loop
 exit when Index > List'Last or else Value = List(Index);

```

When the terminating conditions are met, the problem is solved without further executions of the loop body.

In the general case of the recursive solution, `Value_In_Array` is called to search the remaining, unsearched part of the list. Each recursive execution of the function processes a smaller version of the problem. The smaller-caller question is answered affirmatively because the slice of the array passed is one element smaller on every recursive call. Similarly, in an iterative solution, each subsequent execution of the loop body processes a smaller version of the problem. The unsearched part of the list is shrunk on each execution of the loop body by incrementing `Index`. Here is the iterative version of the function:

```

function Value_In_Array (List : in Array_Type,
 Value : in Integer) return Boolean is
-- Searches List for Value using iteration

-- Preconditions : None

 Index : Integer;
begin
 Index := List'First; -- Start with first value in array
 loop
 exit when Index > List'Last or else Value = List (Index);
 Index := Index + 1;
 end loop;
 return Index <= List'Last and then Value = List (Index);
end Value_In_Array;

```

---

#### **Tail Recursion**

The case where a recursive subprogram contains a single recursive call that is the last statement in the subprogram.

---

Cases where the recursive call is in the last statement executed are called **tail recursion**. Note that the recursive call is not necessarily the last statement in the procedure. For instance, the recursive call in the following version of `Value_In_Array` is still tail recursion, even though it is *not* the last physical statement in the procedure:

```

function Value_In_Array (List : in Array_Type,
 Value : in Integer) return Boolean is
-- Searches List for Value using tail recursion

-- Preconditions : None

 Result : Boolean;
begin
 if List'Length /= 0 then
 if Value = List(List'First) then
 Result := True;
 else
 Result := Value_In_Array -- Search rest of List
 (List => List (List'First+1..List'Last),
 Value => Value);
 end if;
 end if;
 return Result;
end Value_In_Array;

```

```

 end if;
else
 Result := False;
end if;
return Result;
end Value_In_Array;
```

The recursive call is the last statement *executed* in the general case—thus it is tail recursion. Tail recursion is easily replaced by iteration to remove recursion from the solution.

### Stacking

When the recursive call is *not* the last action executed in a recursive procedure, we can't simply substitute a loop for the recursion. For instance, in procedure `Reverse_Traverse` we make the recursive call and then print the value in the current node. In cases like this, we must replace the stacking that was done by the *system* with stacking that is done by the *programmer*.

How would we write procedure `Reverse_Traverse` nonrecursively? As we traverse the list, we must keep track of the pointer to each node, until we reach the end of the list (when our traversing pointer equals `null`). When we reach the end of the list, we process the `Info` field of the last node. Then we back up and process again, back up and process, and so on, until we have printed the first list element.

We know of a data structure in which we can store pointers and retrieve them in reverse order: the stack. The general task for `Reverse_Traverse` is

### Reverse\_Traverse (iterative)

```

-- Construct a stack of pointers (one pointer to each node in the list)
Ptr := pointer to first node in list
loop
 exit when Ptr is null
 Push Ptr onto the stack
 Advance Ptr
end loop
-- Process the nodes in reverse order (last node first)
loop
 exit when Stack is empty
 Pop the stack to get Ptr (to previous node)
 Process (Ptr.all.Info)
end loop
```

A nonrecursive `Reverse_Traverse` procedure may be coded as follows:

```

procedure Reverse_Traverse (List : List_Type) is
-- Prints out the elements in the list in reverse order.
-- This is a nonrecursive procedure.
```

```

-- Instantiate a stack package for stacks of node pointers
package Ptr_Stack is new Stack (Element => Node_Ptr);
-- Local variables
Stack : Ptr_Stack.Stack_Type; -- Stack of pointers
Ptr : Node_Ptr; -- Pointer to current node
```

```

begin
 -- Push pointers to all nodes onto the stack
 Ptr := List; -- Start with first node
 loop -- Each iteration, a pointer to a node is pushed
 exit when Ptr = null; -- Exit at end of list
 -- Push an alias that points to the current element
 Ptr_Stack.Push (Stack => Stack, Element => Ptr);
 Ptr := Ptr.all.Next; -- Advance Ptr to next node
 end loop;
 -- Retrieve pointers in reverse order and process elements
 loop -- Each iteration, a pointer to a node is popped
 exit when Ptr_Stack.Empty (Stack);
 Ptr_Stack.Pop (Stack => Stack, Element => Ptr);
 Process (Ptr.all.Info);
 end loop;
end Reverse_Traverse;

```

Notice that the nonrecursive version of `Reverse_Traverse` is quite a bit longer than its recursive counterpart, especially if we consider the code for the stack package. This verbosity is caused by our need to stack and unstack the pointers explicitly. In the recursive version, we just called `Reverse_Traverse` recursively, and let the run-time stack keep track of the pointers.

### Deciding Whether to Use a Recursive Solution

There are several factors to consider in deciding whether or not to use a recursive solution to a problem. The main issues are the clarity and the efficiency of the solution. Let's talk about efficiency first. In general, a recursive solution is more costly in terms of both computer time and space. (This is not an absolute decree; it really depends on the computer and the compiler.) A recursive solution usually requires more "overhead" because of the nested recursive procedure or function calls, in terms of both time (the procedure prologues and epilogues must be run for each recursive call) and space (an activation record must be created). A call to a recursive routine may hide many layers of internal recursive calls. For instance, the call to an iterative solution to `Factorial` involves a single function invocation, causing one activation record to be put on the run-time stack. Invoking the recursive version of `Factorial`, however, requires  $N + 1$  function calls and  $N + 1$  activation records to be pushed onto the run-time stack. That is, the depth of recursion is  $O(N)$ . For some problems, the system just may not have enough space in the run-time stack to run a recursive solution.

Let's look at the data from two simple experiments. For the first experiment we created a linked list containing 100,000 integers. Then we measured the amount of CPU time it took to traverse this list with both a recursive procedure and an iterative procedure. In the second experiment we created an ordered array-based list containing 100,000 integers. Then we measured the amount of CPU time it took to search for the smallest value in the list—the worst case for a binary search. Because the binary search is so efficient, we repeated the search 10,000 times to obtain the times shown. Here are the results of these experiments:

**Results of the Timed Implementation Comparisons (in seconds)**

| <i>Procedure</i> | <i>Recursive<br/>Implementation</i> | <i>Iterative<br/>Implementation</i> |
|------------------|-------------------------------------|-------------------------------------|
| Traversal        | 1.666                               | 0.114                               |
| Binary Search    | 0.728                               | 0.342                               |

The iterative traversal was nearly 15 times faster than the recursive traversal! This observation clearly demonstrates the amount of overhead of the recursive calls. Although the binary search results again show that the iterative version is faster, the difference is really not great enough to rule out using the recursive version. As usual, keep in mind that these results are only valid for the particular computer and Ada compiler used in the experiments.

Another problem to look for is the possibility that a particular recursive solution might just be *inherently* inefficient. Such inefficiency is not a reflection of how we choose to implement the algorithm; rather, it is an indictment of the algorithm itself. For instance, look back at function `Combinations`, which we discuss earlier in this chapter. The example of this function illustrated in Figure 14-4 [`Combinations (4, 3)`] seems straightforward enough. But consider the execution of `Combinations (6, 4)`, as illustrated in Figure 11-10. The inherent problem with this function is that the same values are calculated over and over. `Combinations (4, 3)` is calculated in two different places, and `Combinations (3, 2)` is calculated in three places, as are `Combinations (2, 1)` and `Combinations (2, 2)`. It is unlikely that we could solve a combinatorial problem of any large size using this procedure. The problem is that the program runs "forever"—or until it exhausts the capacity of the computer. It's an exponential-time [ $O(2^N)$ ] solution to a linear-time [ $O(N)$ ] problem. Although our recursive function is very easy to understand, it isn't a practical solution. In such cases an iterative solution must be sought.

The issue of the clarity of the solution is still a very important factor. For many problems a recursive solution is simpler and more natural for the programmer to write. The total amount of work required to solve a problem can be envisioned as an iceberg. By using recursive programming, the applications programmer may limit his or her view to the tip of the iceberg. The system takes care of the great bulk of the work below the surface. Compare, for example, the recursive and nonrecursive versions of procedure `Reverse_Traverse`. In the recursive version we let the system take care of the stacking that we have to do explicitly in the nonrecursive procedure. Thus recursion is a tool that can help reduce the complexity of a program by hiding some of the implementation details.

To summarize, it is good to use recursion when these are the situations:

- *The depth of recursive calls is relatively "shallow,"* some fraction of the size of the problem. For instance, the level of recursive calls in the `Binary_Search` procedure is  $O(\log_2 N)$ ; this is a good candidate for recursion. The depth of recursive calls in the `Factorial` and `Value_In_Array` routines, however, is  $O(N)$ .
- *The recursive version does about the same amount of work as the nonrecursive version.* You can compare the Big-O approximations to determine this. For instance, we have determined that the  $O(2^N)$  recursive version of `Combinations` is a poor use of recursion, compared to an  $O(N)$  iterative version. Both the recursive and iterative versions of `Binary_Search`, however, are  $O(\log_2 N)$ . The recursive version of `Binary_Search` is a good use of a recursion.

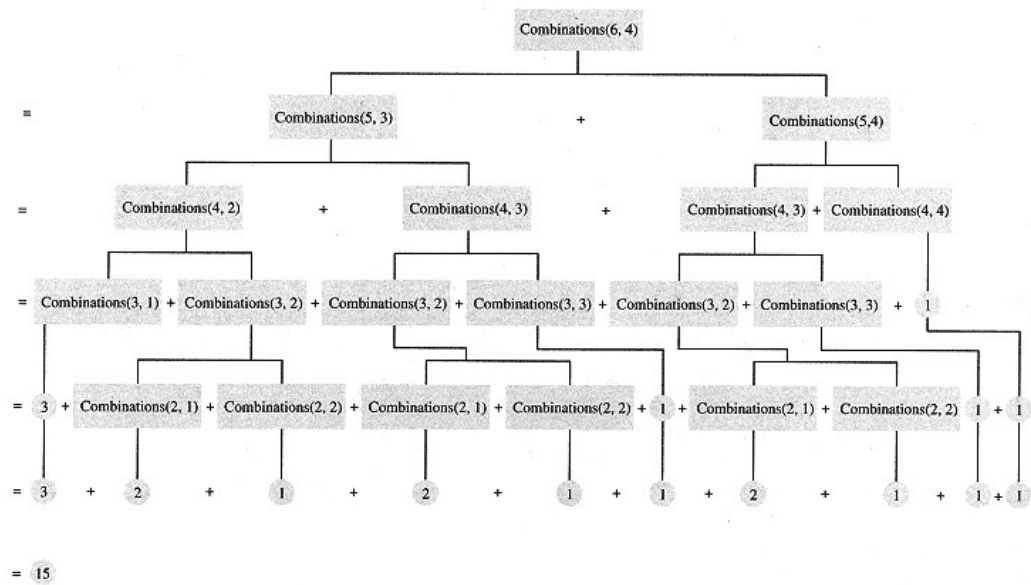


Figure 11-10  
Calculating Combinations (6, 4)

- *The recursive version is shorter and simpler than the nonrecursive solution.* By this rule, `Factorial` and `Value_In_Array` are not good uses of recursive programming. They illustrate how to understand and write recursive procedures and functions, but they could more efficiently be written iteratively—without any loss of clarity in the solution. `Reverse_Traverse` is a better use of recursion. Its recursive solution is very simple to understand, and the nonrecursive equivalent is much less elegant.

In the next chapter we look at an inherently recursive structure where recursive algorithms are much simpler than their iterative counterparts.

The differences between recursive and iterative solutions suggest a good general programming strategy. First, develop and test a clearly understandable recursive solution to a problem. Then convert this recursive solution into a possibly less readable but more efficient iterative one using our two techniques for removing recursion.

### Summary

Recursion is a very powerful computing tool. Used appropriately, recursion can simplify the solution of a problem, often resulting in shorter, more easily understood source code. As usual, there are trade-offs: Recursive procedures are often less efficient, in terms of both time and space, due to the overhead of many levels of procedure calls. How expensive this cost is depends on the computer system and compiler.

A recursive solution to a problem must have at least one base case—that is, a case where the solution is derived nonrecursively. Without a base case, the procedure or function recurses forever (or at least until the computer runs out of memory). The recursive solution also has one or more general cases that include recursive calls to the procedure or function. The recursive calls must involve a "smaller caller." One (or more) of the actual parameter values must change in each recursive call to redefine the problem to be closer to the base case than it was on the previous call. Thus each recursive call leads the solution of the problem toward the base case(s).

A typical implementation of recursion involves the use of a stack. Each call to a subprogram generates an activation record to contain its return address, parameters, and local variables. The activation records are accessed in a Last-In-First-Out manner. Thus a stack is the choice of data structure. Recursion can be supported by systems and languages that use dynamic storage allocation. The procedure parameters and local variables aren't bound to addresses until an activation record is created at run time. Thus multiple copies of the intermediate values of recursive calls to the program can be supported, as new activation records are created for them.

With static storage allocation, in contrast, a single location is reserved at compile time for each parameter and local variable of a procedure. There is no place to store intermediate values calculated by repeated nested calls to the same procedure. Therefore, systems and languages with only static storage allocation can't support recursion.

When recursion isn't possible or appropriate, a recursive algorithm can be implemented nonrecursively by using a looping structure and, in some cases, by pushing and popping relevant values onto our own stack. This programmer-controlled stack explicitly replaces the system's run-time stack. While such nonrecursive solutions are often more efficient in terms of time and space, there is usually a trade-off in terms of the elegance of the solution.

In the case study that follows, we develop a recursive solution to the problem of finding the way out of a maze. This problem was once included in a Pascal version of this book as a stack application. We said earlier that any recursive solution can be implemented using iteration or a stack. Our case study illustrates another way of looking at this idea: any problem that uses a stack in its solution may be a good place to use a recursive algorithm.

### Case Study:

#### Escape from a Maze

As a child, did you ever dream of playing in a maze? How fun and scary it would have been to get lost and then, just at sundown, find your way out. If you had thought about it, you might have come up with the idea of marking your path as you went along. If you were trapped, you could then go back to the last crossing and take the other path.

This technique of going back to the last decision point and trying another way is called *backtracking*. We illustrate this very useful problem-solving technique in the context of trying to get out of a maze.

Given a maze (like the one illustrated on the right side of Figure 11-11) and a starting point within it, you are to determine whether there is a way out. There is only one exit from the maze. You may move horizontally or vertically (but not diagonally) in any direction in which there is an open path, but you may not move in a direction that is blocked. If you move into a position where you are blocked on three sides, you must go back the way you came (backtrack) and try another path.

We write a program that uses this technique to escape from a maze, according to the following specifications:

### Specifications:

#### Program Escape

#### Function

This program uses a backtracking technique to escape from a maze.

#### Input

The input comes from two text files. The first, `Maze_File`, contains the original maze, represented as a square matrix of symbols, with one row of the matrix per input line. Each symbol is a character that indicates whether the corresponding maze position is an open path ('O'), a trap ('+'), or the finish ('F'). The size of the maze is 10 by 10 positions. Here is an example of the data that we might find in this file:

```
OO+F+OO++
O++O+O+OOO
OOOOOO+O+O
+++++O++OO
OOO+OOO+O+
O+O+O+++O+
O+O+OOO+OO
++O+++O++O
O+OOOOO+O
O+O++O+OOO
```



The second file, `Start_File`, contains a series of starting position coordinates. Each line contains a pair of values, representing the row and column of the starting position in the maze. You are to process each of these starting points until the end of the file is reached.

**Output**

For each starting position, print the following:

- 1. The maze with the starting point represented by a \* symbol.
- 2. A message that indicates the result of the escape, "HOORAY! I am free!" or "HELP! I am trapped!"
- 3. The number of positions tried before a solution was found.

**Processing Requirements**

Begin processing each starting position at the specified coordinates, and continue moving until you find the way out or have no more moves to try. You may move horizontally or vertically into any position that is an open path ('O'), but not into any position that is blocked ('+'). If you move into the finish position, ('F'), you have exited the maze. (Don't worry if the number of positions tried is greater than the number of positions in the maze; there may be positions that were duplicated during backtracking.)

**Assumptions**

The two input files exist; all input is valid.

The starting position is not marked as a trap.

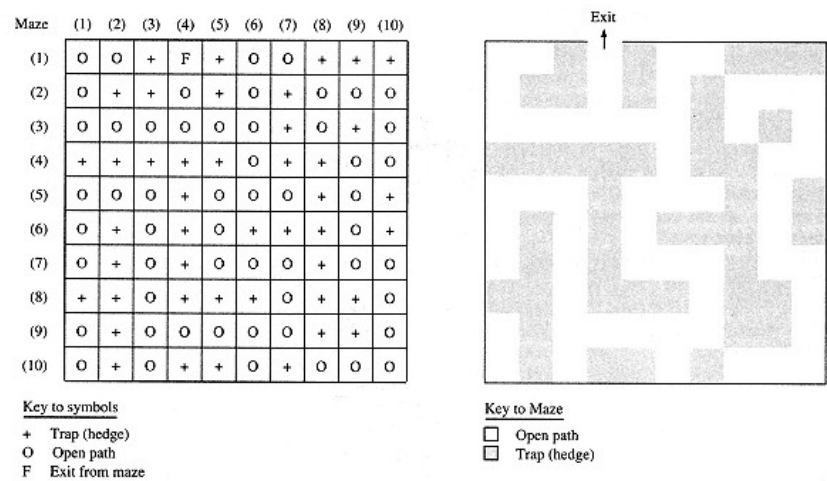


Figure 11-11  
Picture of a Maze

## The Design

The maze is the major object in this problem. Because we only need one maze, we model it as an ADO. The maze ADO requires operations to (1) mark a position on the maze and (2) return the "symbol" value of a maze position.

From the logical perspective, the maze itself can be thought of as a two-dimensional grid of "positions" in the maze. Each position can be identified by its Row and Column numbers—values between 1 and 10, according to the specifications. Each position contains a Symbol that represents its contents: Open (an open path), Trap (a dead end), Finish (exit from the maze), or Start (the starting point for the escape attempt). The specifications tell us that the file input designates an open path by 'O', a trap by '+', and the finish by 'F'. When we print the maze, the starting point is to be marked with the character '\*'. For convenience, in our illustrations, we use these characters to represent the contents of the maze positions. Figure 11-11 shows the logical picture of a maze.

The two maze operations can be specified as follows:

```
procedure Mark (Row : in Row_Type;
 Column : in Col_Type;
 Symbol : in Symbol_Type);
-- Set the maze position of the specified Row and Column
-- to the value of symbol

-- Preconditions: None

-- Postconditions: Maze is updated to include modified Symbol value
-- at the position specified by Row and Column

function Symbol_At (Row : in Row_Type;
 Column : in Col_Type) return Symbol_Type;
-- Returns the Symbol value of the maze position at the specified
-- Row and Column

-- Preconditions: None

-- Postconditions: Symbol_At = symbol value of the maze position at
-- Row and Column
```

Now that the maze operations have been fully specified, let's turn our attention to the overview of the processing with a Level 0 design.

## Escape

### Level 0

```
Get Maze from Maze_File
loop
 exit when end of Start_File
 Get Start Point from Start_File
 Display Maze
 Try to Escape
 Display Results
end loop
```

Using our Maze ADO operations, we can describe much of the program's processing. In *Get Maze from File*, we input the original maze from `Maze_File` by reading the values, row by row, and calling `Mark` to put the appropriate symbols (Open, Trap, or Finish) into the specified positions in the maze. In *Get Start Point*, we mark a starting point by calling `Mark` to set the designated maze position to the Start symbol. *Display Maze* displays the maze row by row, using the `Symbol_At` function to retrieve the symbol value of each maze position. Displaying the results of the escape attempt merely involves a call or two to `Text_IO.Put_Line`. The Level 1 designs for *Get Maze from File*, *Get Start Point*, *Display Maze*, and *Display Results* are straightforward, so let's go on to the interesting part of the problem: *how do we get out of the maze?*

### Escape Processing

Let's look more closely at the *Try To Escape* operation. It needs to know the position (Row and Column) of the starting point. It does some processing and then returns the results: whether the escape was successful and a count of how many moves we had to make in the escape attempt. Here is the procedure's specification:

```
procedure Try_To_Escape (Start_Row : in Maze.Row_Type;
 Start_Column : in Maze.Column_Type;
 Free : in out Boolean;
 Tries : in out Natural);
-- Given a starting position, determine if there is an escape path and
-- determine how many positions have to be tried during the escape.

-- Preconditions: The Maze ADO has been initialized.

-- Postconditions: Free indicates whether an escape path was found;
-- Tries = number of positions tried during the escape.
-- (A position may be tried more than once because of
-- backtracking.)
```

What is this escape processing? Beginning at the starting point, we can move in any direction—up, down, right, or left—but only to an Open path ('O') or the Finish ('F'). Given the part of the maze shown in Figure 11-12, this means that we could move up or left, into an Open path. However, we cannot move down or right, where there are Traps ('+').

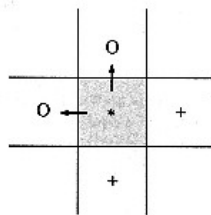


Figure 11-12  
The Starting Position

Let's say that we move to the left. Now we are facing the same kind of situation (see Figure 11-13). We can move in any direction where we are not trapped. Because the positions above and below the current position are blocked, we do not try these. To the right is the Start position (\*). There's no use going back there, so we move again to the left.

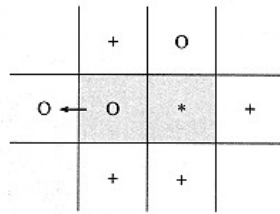


Figure 11-13  
After the First Move

Again we are in the same kind of situation (see Figure 11-14). We can move in any direction that is not blocked. There is a Trap to the left, so we don't move there. There are Open positions above and below and to the right. We just came from the position to the right, so it's not worth going back there. In fact, this is the same situation we faced in Figure 11-13, when we decided not to go back into the Start position. That time, because we saw the Start symbol in the maze position, we knew not to move into the spot. This time, however, it is just another Open position. How do we know not to go back?

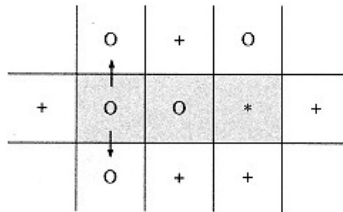


Figure 11-14  
After the Second Move

If you were hiking in an uncharted area, you might mark your trail by breaking twigs or leaving stones in the path you had traveled. We can use that same idea here. We mark the trail of positions that we've already tried by replacing the Open symbol with a special symbol, Tried. (We use a little stone ['.'] to depict the Tried symbol.)

We have two choices in Figure 11-14: up or down. We decide to go up (see Figure 11-15). Now we face the usual choice with a modified restriction. We can go in any direction that is not blocked or *already tried*. Uh-oh! Our new position is blocked in all the untried directions. We are trapped.

But we don't know for sure that there is no way to get out of the maze. We only know that the path we tried doesn't get us there. We passed up a couple of Open positions (see Figures 11-12 and 11-14) as we were going along the path that led us into the dead end. We could go back to one of them and try another path from there. Even better, we can collect *all* these alternative paths as we pass them; then if we get trapped we can go back to the path most recently passed and resume the escape attempt.

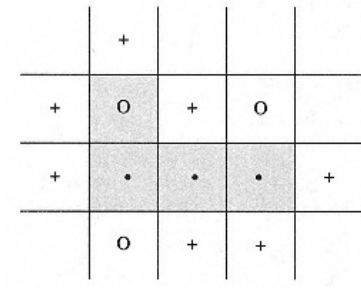


Figure 11-15  
After the Third Move, Marking Our Trail

### A Stack-Based Solution

This line of thinking leads us to the idea of using a data structure that we are already familiar with—the stack—to collect the alternative moves. Every time we enter a position, we put all the possible alternatives (Open positions adjacent to the current position) onto a stack. Then to move, we pop the top alternative off the stack. This move takes us to a new position, from which we push all of the alternative moves onto the stack, and so on. The following algorithm describes this solution:

#### Try To Escape – Stack Version

```

Tries := 0
loop
 Increment Tries
 Push the positions (Row and Column) of all possible moves
 (up, down, left, or right positions that are marked
 as Open or Finish) onto Stack
 exit when Current Maze Position = Finish or the Stack is Empty
 Mark current position as Tried
 Pop new Current Position off the stack
end loop;
Free := Current Maze Position = Finish

```

### A Recursive Solution

This stack-based solution sounds as if it would work, but before we actually try it we notice something else in the discussion. Every time we make a move, we say, "Now we are facing the same situation . . ." Every time we move into a new position, we face a version of the same problem that led us to call `Try_To_Escape` in the first place. This suggests a recursive solution to the problem; perhaps we can use recursion to let the system take care of the backtracking, instead of using a stack.

Let's look at the *general case* of a recursive version of the `Try_To_Escape` procedure. The general case is when the current position, on entering the procedure, is Open. Our current position is described by Row and Column numbers. What are the recursive calls that would solve the problem of escaping from the maze? We could `Try_To_Escape` up or down or right or left. Actually, we could `Try_To_Escape` in *all* of these directions. Here's a first pass at a recursive algorithm for `Try_To_Escape`:

### Try To Escape – Recursive Version 1

```

Increment Tries
if base case then
 Do something nonrecursive
else (general case)
 Mark current position as Tried.
 Try_To_Escape (Row + 1, Column, Free, Tries)
 Try_To_Escape (Row - 1, Column, Free, Tries)
 Try_To_Escape (Row, Column + 1, Free, Tries)
 Try_To_Escape (Row, Column - 1, Free, Tries)
end if

```

What is the base case? This is the case that can be solved nonrecursively, ensuring that the procedure eventually ends. The situations we might face when we enter procedure `Try_To_Escape` are summarized in this table:

#### Try To Escape Processing

| <i>If the content of<br/>Current Position is</i> | <i>Then Try_To_Escape should</i>             |
|--------------------------------------------------|----------------------------------------------|
| Tried                                            | Do nothing (return from procedure)           |
| Trap                                             | Do nothing (return from procedure)           |
| Finish                                           | Set Free to True (and return from procedure) |
| Open (or Start)                                  | Execute general case                         |

In addition, because a call to `Try_To_Escape` may cause `Free` to be set to `True`, a subsequent call may be made with `Free` already `True`. In this case we don't want to do anything except exit the procedure. Let's summarize these conditions in a second draft of the `Try_To_Escape` algorithm.

### Try To Escape – Recursive Version 2

```

Increment Tries
Current_Position := Maze.Symbol_At (Row, Col)
if Free then
 Do Nothing -- Base case, problem already solved
elsif Current_Position = Tried or Current_Position = Trap then
 Do Nothing -- Base case, can't go this way
elsif Current_Position = Finish then
 Free := True -- Base case, found the exit
else -- General case, Current Position is Open
 -- mark this position so we don't try it again
 Maze.Mark (Row, Column, Tried)
 -- try all adjacent positions
 Try_To_Escape (Row + 1, Column, Free, Tries)
 Try_To_Escape (Row - 1, Column, Free, Tries)
 Try_To_Escape (Row, Column + 1, Free, Tries)
 Try_To_Escape (Row, Column - 1, Free, Tries)
end if;

```

## Verifying the Recursive Design

Let's use our three questions to verify that this recursive procedure works.

1. *The Base-Case Question:* The base cases occur when the current position is not an Open path, in which case we should stop searching this path (do nothing), and when Free is already True on entrance to the procedure, in which case we already know the answer to the problem and can quit.

2. *The Smaller-Caller Question:* We need to show here that the portion of the maze left to be processed is smaller on each recursive call. It is only the Open positions that are processed. Each time we set the value of an Open position to Tried, the size of the maze that is left to process becomes "smaller" by one. We usually look at the parameters of the recursive calls to see how the size of the problem is changing. In this case, the maze is not a parameter; it is a globally accessible ADO.

For each recursive call, the Row and Column parameters are adjusted to guide the search for the finish into new paths adjacent to the current position. This observation, however, leads us to note a shortcoming of our design: As we increment and decrement the Row and Column parameters, what keeps us from going right off the edge of the maze? The design needs to be modified to account for the edges of the maze. We finish our proof and then come back to this problem.

3. *The General-Case Question:* Let's assume that a call to `Try_To_Escape` correctly tells us whether an escape can be made from a given position. If we can escape from some known position in the maze, then we can also escape from an Open position adjacent to that position. So if we check all the positions adjacent to the starting point (the four calls to `Try_To_Escape`) and find that any of them leads to an escape, then we know that we can escape from the starting point. The converse is also true: If none of the adjacent positions leads to the finish, then the starting point cannot lead to the finish. So the general case should solve the problem, assuming that the recursive calls do what they're supposed to do.

## The Boundary Problem

We noticed during our verification of the smaller-caller question that there isn't anything to keep us from going over the edges of the maze as Row and Column are incremented and decremented in the recursive calls. We can add explicit checks for the lower and upper bounds of the array index:

```
if (Row < 1) or (Row > 10) or (Column < 1) or (Column > 10) then
 Do nothing, just exit
```

Another way to handle this condition is to modify the data structure that supports the maze so that the existing conditions take care of the problem. We can add a "hedge" that borders the whole maze (another row on the top and bottom of the maze, and another column on the left side and right side of the maze), setting all of these border positions to the Trap symbol. Then if the current position is in the border, it is processed the same way as any other Trap position, that is, we just exit the recursive call to the procedure.

When do we add the boundary hedges to the maze? The specification does not include the Trap symbols for the boundary in the text file. We can make use of package initialization code to set up the hedges in the Maze ADO.

## Finishing the Design

Let's review and refine the top level of our program and then complete the design. There are only a couple of changes to be made in the top level. Because `Try_To_Escape` is a recursive procedure, `Free` and `Tries` must be initialized before the initial call. If we initialize them at the beginning of the recursive procedure, they are reinitialized with every recursive call. Also, because we decided to mark our trail through the maze by changing some of its positions from Open to Tried symbols, we had better use a fresh copy of the original maze each time we process a new starting point. We could implement the Maze as an ADT rather than an ADO. Then we can keep the original maze that we read from the file in a variable called `Original_Maze`, and copy it into a "working" variable called `Maze` before we do any other processing inside the main loop. A second method is to reread the data from the maze file to reset the maze. We use this second method in our solution.

### Escape

*Level 0*

```

loop
 Get Maze From Maze_File
 exit when end of Start_File
 Get Start Point (Row and Column) from Start_File
 Display Maze
 Free := False
 Tries := 0
 Try to Escape (Row, Column, Free, Tries)
 Display Results
end loop

```

Let's move on now to the specification of the Maze ADO. We specified two operations that must be included. We must also declare the types of the operations' parameters. To prevent any confusion between Row and Column numbers, we should use a different type for each. There are four symbols used in the maze (Open, Trap, Finish, or Tried) represented by their single-character equivalents. Declaring constants for these characters helps make our program easier to read. We can use package initialization code to give the maze an initial value. By assigning Trap to all positions, we also set up the boundary hedges. The declaration of our Maze ADO is given in Specification 11-1.

### Specification 11-1— Maze ADO

```

package Maze is

-- This ADO models a maze. Initially, all maze positions are marked as Trap.

 subtype Symbol_Type is Character;

 Open : constant Symbol_Type := 'O';
 Trap : constant Symbol_Type := '+';
 Finish : constant Symbol_Type := 'F';
 Start : constant Symbol_Type := '*';
 Tried : constant Symbol_Type := '.';

```



```

SYMBOL_ERROR : exception; -- Raised if some character other
 -- than the four above is used

Maze_Size : constant := 10;

-- Types for maze array
type Column_Range is range 0..Maze_Size + 1;
type Row_Range is range 0..Maze_Size + 1;

-- Subtypes for positions inside the boundary hedge
subtype Internal_Column is Column_Range range 1..Maze_Size;
subtype Internal_Row is Row_Range range 1..Maze_Size;

procedure Mark (Row : in Internal_Row;
 Column : in Internal_Column;
 Symbol : in Symbol_Type);
-- Set the maze position of the specified Row and Column to the value of symbol

-- Preconditions: None

-- Postconditions: Maze is updated to include modified Symbol value
-- at the position specified by Row and Column

-- Exceptions: SYMBOL_ERROR if Symbol is not a legal symbol for the maze

function Symbol_At (Row : in Row_Range;
 Column : in Column_Range) return Symbol_Type;
-- Returns the Symbol value of the maze position at the specified Row and Column
-- Includes the bounding hedge.

-- Preconditions: None

-- Postconditions: Symbol_At = symbol value of the maze position at Row and Column
end Maze;

```

We have already discussed the design of `Try_To_Escape`. With the specification of the Maze ADO complete, the remaining steps in our algorithm, `Get_Maze`, `Get_Start_Point`, `Display_Maze`, and `Display_Results`, are trivial and can be coded without further design with the program specification as a guide.

### ***The Implementation***

We've managed to get all the way through the design without tying ourselves down to a representation of the maze. Although we could probably think up more interesting structures to implement the maze, the most natural is a two-dimensional array of symbols. Using this representation, each maze operation takes only a single assignment statement. Is it worth encapsulating these one-liners in an ADO instead of just referencing an array in the calling program? There are several reasons for using the ADO. We could change the implementation (say, to a linked list) with no change in the rest of the program. All of the

related types (Row\_Type, Column\_Type, and Symbol\_Type) are together. And we can use package initialization code to set up the borders of the maze. Body 11-1 contains the body of the Maze ADO, and the code for our program is shown in Program 11-1.

#### Body 11-1—

#### Maze ADO Implements Specification 11-1

```
package body Maze is

 type Maze_Type is array (Row_Range, Column_Range) of Symbol_Type;

 The_Maze : Maze_Type; -- The maze "object"

 procedure Mark (Row : in Internal_Row;
 Column : in Internal_Column;
 Symbol : in Symbol_Type) is
 begin
 if Symbol = Trap or Symbol = Open or Symbol = Tried or
 Symbol = Finish or Symbol = Start then
 -- Mark the maze
 The_Maze (Row, Column) := Symbol;
 else
 raise SYMBOL_ERROR;
 end if;
 end Mark;

 function Symbol_At (Row : in Row_Range;
 Column : in Column_Range) return Symbol_Type is
 begin
 return The_Maze (Row, Column) ;
 end Symbol_At;

begin -- package initialization
 -- Initialize entire maze to traps
 The_Maze := (Row_Range => (Column_Range => Trap));
end Maze;
```

#### Program 11-1—

#### Escape

```
with Integer_IO;
with Maze;
with Text_IO;
procedure Escape is
```

```

-- Instantiate I/O packages
package Column_IO is new Text_IO.Integer_IO (Num => Maze.Internal_Column);
package Row_IO is new Text_IO.Integer_IO (Num => Maze.Internal_Row);

Maze_File : Text_IO.File_Type; -- File containing the maze
Start_File : Text_IO.File_Type; -- File containing starting positions

Column : Maze.Internal_Column; -- Maze position
Row : Maze.Internal_Row; -- variables

Free : Boolean; -- Result
Tries : Natural; -- Number of tries to escape

procedure Get_Maze (Maze_File : in Text_IO.File_Type) is
-- Gets values for maze positions from Maze_File. Each line of this file
-- contains one row of the maze. Data is assumed to be valid.

-- Preconditions : Maze_File is open and the reading marker is
-- positioned at the beginning of the file.

 Symbol : Maze.Symbol_Type; -- A symbol for one maze position

begin
 for Row in Maze.Internal_Row loop
 for Column in Maze.Internal_Column loop
 Text_IO.Get (File => Maze_File, Item => Symbol);
 Maze.Mark (Row, Column, Symbol);
 end loop;
 Text_IO.Skip_Line (Maze_File); -- Go to next line of file
 end loop;
end Get_Maze;

procedure Get_Start_Point (Start_File : in Text_IO.File_Type;
 Row : out Maze.Internal_Row;
 Column : out Maze.Internal_Column) is
-- Gets the starting point from file and marks it on the maze

Local_Row : Maze.Internal_Row; -- Local copies of
Local_Col : Maze.Internal_Column; -- out parameters

begin
 Row_IO.Get (File => Start_File, Item => Local_Row);
 Column_IO.Get (File => Start_File, Item => Local_Col);
 Maze.Mark (Row => Local_Row, Column => Local_Col, Symbol => Maze.Start);
 Row := Local_Row;
 Column := Local_Col;
end Get_Start_Point;

```

```

procedure Display_Maze is
-- Displays the maze
begin
 Text_IO.New_Page;
 for Row in Maze.Internal_Row loop
 for Column in Maze.Internal_Column loop
 Text_IO.Put (Item => Maze.Symbol_At (Row, Column));
 end loop;
 Text_IO.New_Line;
 end loop;
end Display_Maze;

procedure Try_To_Escape (Row : in Maze.Row_Range; -- Must include the
 Column : in Maze.Column_Range; -- enclosing hedge
 Free : in out Boolean;
 Tries : in out Natural) is
use Maze; -- To make the + and - operators for Row and Column directly visible
begin
 Tries := Tries + 1;
 if not Free then -- Do nothing if already free (Base case)
 case Maze.Symbol_At (Row, Column) is
 when Maze.Finish => -- Base case that requires an action
 Free := True;
 when Maze.Trap | -- Remaining base cases require no action
 Maze.Tried =>
 null;
 when Maze.Open |
 Maze.Start => -- General case
 Maze.Mark (Row, Column, Maze.Tried);
 Try_To_Escape (Row + 1, Column, Free, Tries);
 Try_To_Escape (Row - 1, Column, Free, Tries);
 Try_To_Escape (Row, Column + 1, Free, Tries);
 Try_To_Escape (Row, Column - 1, Free, Tries);
 when others => -- Invalid symbol
 raise Maze.SYMBOL_ERROR;
 end case;
 end if;
end Try_To_Escape;

procedure Display_Results (Free : in Boolean; Tries : in Natural) is
-- Display the results of the escape attempt
begin
 if Free then
 Text_IO.Put_Line ("HORRAY! I am free!");
 else
 Text_IO.Put_Line ("Help! I am trapped!");
 end if;
 Text_IO.Put ("Number of moves taken = ");
 Integer_IO.Put (Item => Tries, Width => 1);
 Text_IO.New_Line;
end Display_Results;

```

```

begin -- Escape
 Text_IO.Open (File => Maze_File,
 Mode => Text_IO.In_File,
 Name => "Maze.Dat");
 Text_IO.Open (File => Start_File,
 Mode => Text_IO.In_File,
 Name => "Start.Dat");
 loop -- Each iteration, an attempt is made to escape from the maze
 exit when Text_IO.End_Of_File (Start_File);
 Text_IO.Reset (Maze_File); -- Set reading marker to beginning of file
 Get_Maze (Maze_File); -- Get the initial maze (unmarked by Tried symbols)
 Get_Start_Point (Start_File, Row, Column);
 Display_Maze;
 Free := False; -- Initialize parameters
 Tries := 0; -- for recursive call
 Try_To_Escape (Row, Column, Free, Tries);
 Display_Results (Free, Tries);
 end loop;
 Text_IO.Close (Maze_File);
 Text_IO.Close (Start_File);
end Escape;

```

### Testing the Program

In our first attempt at testing this program, we try out the input and output routines (`Get_Maze`, `Get_Start_Point`, `Display_Maze`) with sample files of data, commenting out the call to `Try_To_Escape` in the main program. When we see that those procedures work correctly, we get up our nerve to try running the program with the recursive procedure `Try_To_Escape` restored. To make sure that we can trace its execution through all the recursive calls, we put in debug Puts at the start of `Try_To_Escape`, before any other statements:

```

Text_IO.Put ("Try_To_Escape entered with Row ");
Row_IO.Put (Item => Row, Width => 1);
Text_IO.Put (" and Column ");
Column_IO.Put (Item => Column, Width => 1);
Text_IO.Put_Line (" and Free " & Boolean'Image(Free));

```

Then we run the program using a variety of starting positions. The debugging output helps us to check whether the procedure is really solving the problem as we had expected. It does seem to work, but we are really surprised at how many recursive calls have to be made. Sometimes `Tries` is larger than the total number of positions in the maze! But we shouldn't be surprised because we can come back to a position more than once, only to find that we have already been there (in these cases we exit immediately).

Looking through the debugging output, we realize that some of the calls are being made to `Try_To_Escape` even after we have determined that we are Free. For instance, if `Free` is set to True on the first of the four recursive calls in the general case, we still make the other three recursive calls, exiting immediately because `Free` is True (one of the

base cases). We can avoid these extra calls by putting a condition *before* the recursive calls in the general case rather than as a base case *within* Try\_To\_Escape. Then we only make the call if not Free.

```
-- in the general case
.
.
.
Try_To_Escape (to the position above);
if not Free then
 Try_To_Escape (to the position below);
end if;
if not Free then
 Try_To_Escape (to the position on the right);
end if;
if not Free then
 Try_To_Escape (to the position on the left);
end if;
```

We make this change to the Try\_To\_Escape procedure and run the program again on the same input file. This time some (but not all) of the values of Tries are smaller. A comparison of the output values for the two versions of Try\_To\_Escape is shown in the next table. Note that if there is no escape, it doesn't help to make the calls to Try\_To\_Escape conditional—Free is never True. If we do escape, however, there is a noticeable difference between the two versions.

**A Comparison of Output of the Two Versions of Procedure Try\_To\_Escape**

| Starting Coordinates |        | Result  | Number of Moves to Determine Results |           |
|----------------------|--------|---------|--------------------------------------|-----------|
| Row                  | Column |         | Version 1                            | Version 2 |
| 1                    | 2      | Freed   | 33                                   | 20        |
| 10                   | 1      | Trapped | 9                                    | 9         |
| 10                   | 8      | Trapped | 65                                   | 65        |
| 7                    | 6      | Freed   | 121                                  | 108       |
| 1                    | 7      | Freed   | 125                                  | 115       |
| 8                    | 7      | Freed   | 121                                  | 106       |
| 7                    | 9      | Trapped | 65                                   | 65        |
| 9                    | 3      | Freed   | 121                                  | 100       |
| 7                    | 1      | Freed   | 121                                  | 82        |
| 2                    | 8      | Trapped | 65                                   | 65        |

### ***The Recursive Solution Versus a Nonrecursive Solution***

Well . . . was it worth recursing?

One of the techniques we mentioned for deciding whether to use a recursive solution to a problem was to ask if the recursive solution produced a shorter and simpler program. It just so happens that there's a nonrecursive solution to this same program; it uses the stacking algorithm discussed earlier to handle the backtracking.

Not counting the code for the stack package, the two versions have nearly the same number of executable lines of code. Of course, including the code from either of the stack packages we developed in Chapter 6, the stack version makes the total size of the nonrecursive version considerably larger. The nonrecursive version is also less clear; it takes some time to become convinced that all the pushing and popping really solves the problem of getting out of the maze. In this case we believe that the recursive solution is a good choice.

## Exercises

1. Explain what is meant by

- (a) base case
- (b) general (or recursive) case
- (c) run-time stack
- (d) binding time
- (e) tail recursion
- (f) binding

2. *True or False?* If you answer false, correct the statement.

Recursive procedures and functions

- (a) often have fewer local variables than the equivalent nonrecursive routines.
- (b) generally use `loop`, `while`, or `for` statements as their main control structure.
- (c) are possible only in languages with static storage allocation.
- (d) should be used whenever execution speed is critical.
- (e) are always shorter and clearer than the equivalent nonrecursive routines.
- (f) must always contain a path that does not contain a recursive call.
- (g) are always less efficient, in terms of Big-O.

3. Use the Three-Question Method to verify the `Value_In_Array` function in this chapter.

4. Describe the Three-Question Method of verifying recursive routines in relation to an inductive proof.

5. What data structure would you most likely see in a nonrecursive implementation of a recursive algorithm?

6. Using the recursive procedure `Reverse_Traverse`, as a model, write the recursive procedure `Traverse` which traverses the elements in the list in order. Does one of these routines constitute a better use of recursion? If so, which one?

Use the following function in answering Exercises 7 and 8:

```
function Puzzle (Base : in Natural;
 Limit : in Natural) return Integer is
 Result : Integer;
begin -- Puzzle
 if Base > Limit then
 Result := -1;
 elsif Base = Limit then
 Result := 1;
 else
 Result := Base * Puzzle (Base + 1, Limit);
 end if;
 return Result;
end Puzzle;
```

7. Identify the following:

- (a) the base case(s) of function `Puzzle`
- (b) the general case(s) of function `Puzzle`

8. Show what would be written by the following calls to the recursive function `Puzzle`:

- (a) `Integer_IO.Put (Puzzle (14, 10));`
- (b) `Integer_IO.Put (Puzzle (4, 7));`
- (c) `Integer_IO.Put (Puzzle (0, 0));`

9. You are given the following function:

```
function Func (Num : in Num_Type) return Num_Type is
 Result : Num_Type;
begin
 if Num = 0 then
 Result := 0;
 else
 Result := Num + Func (Num + 1);
 end if;
 return Result;
end Func;
```

- (a) Write a type declaration of `Num_Type` that allows this function to pass the smaller-caller test.
- (b) Is `Func (7)` a legal call, given your declaration of `Num_Type`? If so, what is returned from the function?
- (c) Is `Func (0)` a legal call, given your declaration of `Num_Type`? If so, what is returned from the function?
- (d) Is `Func (-5)` a legal call, given your declaration of `Num_Type`? If so, what is returned from the function?
- (e) Is `Func (-5.2)` a legal call, given your declaration of `Num_Type`? If so, what is returned from the function?

10. Put comments on the following routines to identify the base and general cases and explain what each routine does.

```
(a) function Power (Base : in Integer;
 Exponent : in Natural) return Integer is
 Result : Integer;
begin
 if Exponent = 0 then
 Result := 1;
 else
 Result := Base * Power (Base, Exponent - 1);
 end if;
end Power;

(b) function Factorial (Num : in Integer) return Integer is
 Result : Integer;
begin
 if Num > 0 then
 Result := Num * Factorial (Num - 1);
 elsif Num = 0 then
 Result := 1;
 else
 raise CONSTRAINT_ERROR;
 end if;
end Factorial;
```



```

(c) procedure Sort (Data : in out Array_Type) is
 Maxi : Index_Type;
begin
 if Data'Length > 1 then

 -- Find index of the largest element in array
 Maxi := Max_Position (Data);
 -- Exchange the values of the two array elements
 Swap (Data(Maxi), Data(Data'Last));

 Sort (Data => Data (Data'First .. Data'Last - 1));

 end if;
end; (* Sort *)

```

11. (a) Fill in the blanks to complete the function:

```

function Sum (Data : in Array_Type) return Integer is
-- Returns the sum of all the elements in the array Data
 Result : Integer;
begin
 if _____ then
 Result := _____; -- Base case
 else -- General case
 Result := _____;
 end if;
 return Result;
end Sum;

```

(b) Show how you would call this function to sum all the elements in an array called `Numbers`, which contains elements indexed from 1 to `Max_Array`.

(c) What run-time problem might you have with this function as it is coded?

12. Here are the EBNF definitions necessary to define a postfix expression (discussed in Chapter 6). What is the base case?

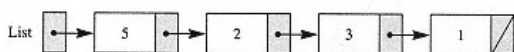
postfix\_expression ::= operand operand operator

operator ::= + | - | \* | /

operand ::= postfix\_expression | integer

13. You must assign the grades for a programming class. Right now the class is studying recursion, and they have been given this simple assignment: Write a recursive procedure `Sum_Squares` that takes a pointer to a linked list of integer elements and returns the sum of the squares of the elements.

*Example:*



$\text{Sum\_Squares}(\text{List}) = (5 * 5) + (2 * 2) + (3 * 3) + (1 * 1) = 39$

Assume that the list is not empty.

You have received quite a variety of solutions. Grade the procedures below, marking errors where you see them.

(a)

```

function Sum_Squares (List : in List_Type) return Natural is
 Sum : Natural;
begin
 Sum := 0;
 if List /= null then
 Sum := (List.all.Info) ** 2 + Sum_Squares (List.all.Next);
 end if;
 return Sum;
end Sum_Squares;

```

(b)

```

function Sum_Squares (List : in List_Type) return Natural is
 Sum : Natural;
 Ptr : List_Type;
begin
 Sum := 0;
 Ptr := List;
 loop
 exit when Ptr = null;
 Sum := Sum + (Ptr.all.Info) ** 2;
 Ptr := Ptr.all.Next;
 end loop;
 return Sum;
end Sum_Squares;

```

(c)

```

function Sum_Squares (List : in List_Type) return Natural is
 Sum : Natural;
begin
 if List = null then
 Sum := 0;
 else
 Sum := (List.all.Info) ** 2 + Sum_Squares (List.all.Next);
 end if;
 return Sum;
end Sum_Squares;

```

(d)

```

function Sum_Squares (List : in List_Type) return Natural is
 Sum : Natural;
begin
 if List.all.Next = null then
 Sum := (List.all.Info) ** 2;
 else
 Sum := (List.all.Info) ** 2 + Sum_Squares (List.all.Next);
 end if;
 return Sum;
end Sum_Squares;

```

(e)

```

function Sum_Squares (List : in List_Type) return Natural is
 Sum : Natural;
begin
 if List = null then
 Sum := 0
 else
 Sum := Sum_Squares (List.all.Next) *
 Sum_Squares (List.all.Next);
 end if;
 return Sum;
end Sum_Squares;

```

14. (a) Using the recursive function on page 481, calculate the value of `Fibonacci (6)` by hand. What does this exercise tell you about the efficiency of the recursive solution? Would you call this a good use of recursion?

(b) Write a nonrecursive version of function `Fibonacci`.

(c) Write a driver to test the recursive and iterative versions of function `Fibonacci`.

15. The following is the declaration of a function that calculates an approximation of the square root of `Num`, starting with an approximate answer (`Approx`), within the specified tolerance (`Tol`):

```

function Sqr_Root (Num : in Float_Type;
 Approx : in Float_Type;
 Tol : in Float_Type) return Float_Type;
-- This function calculates an approximation of the
-- square root of Num.

-- Approx is an initial approximation of the square root of Num
-- Tol is the desired tolerance of the result.

```

The function uses Newton's method, as defined below:

```

Sqr_Root (Num, Approx, Tol) =
 { Approx, if |Approx2 - Num| ≤ Tol
 { Sqr_Root (Num, (Approx2 + Num) / (2*Approx), Tol), if |Approx2 - Num| > Tol

```

(a) What limitations must be made on the values of the parameters, if this method is to work correctly? Write the type declaration for `Float_Type`.

(b) Write a recursive version of function `Sqr_Root`.

(c) Write a nonrecursive version of function `Sqr_Root`.

(d) Write a driver to test the recursive and iterative versions of function `Sqr_Root`.

16. A sequential search procedure has the following declaration:

```

procedure Search (List : in List_Type; -- list to search
 Value : in Key_Type; -- value to search for
 Location : out Loc_Type; -- location of value
 Found : out Boolean); -- search successful?

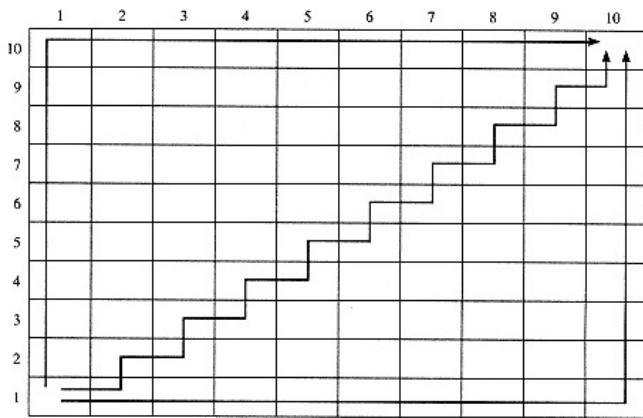
```

(a) Complete the procedure as a recursive search, assuming that `List` is the external pointer to a linked list of elements ordered by their key field, from smallest to largest (use the linked list declarations from Chapter 8). `Loc_Type` is a pointer to a list node. If not `Found`, `Location` is undefined.

(b) Complete the procedure as a recursive search, assuming that `List_Type` is an unconstrained array of `Key_Type`. The list elements are ordered from smallest to largest. `Loc_Type` is an array index. If `Value` is not Found in the list, `Location` is undefined. You cannot change the parameter list of `Search`.

(c) Write drivers to test your two procedures.

17. We want to count the number of paths possible to move in a two-dimensional grid from row 1, column 1 to row  $N$ , column  $N$ . Steps are restricted to going up or to the right, but not diagonally. The illustration below shows three of many paths, if  $N = 10$ :



(a) The following function, `Num_Paths`, is supposed to count the number of paths, but it has some problems. Debug the function.

```
function Num_Paths (Row : in Positive; -- Starting Row
 Col : in Positive; -- Starting Column
 N : in Positive) return Positive is
 Result : Positive;
begin
 if Row = N then
 Result := 1;
 elsif Col = N then
 Result := Num_Paths + 1;
 else
 Result := Num_Paths (Row + 1, Col) *
 Num_Paths (Row, Col + 1);
 end if;
 return Result;
end Num_Paths;
```

(b) After you have corrected the function, trace the execution of `Num_Paths` with  $N = 4$  by hand. Why is this algorithm inefficient?

(c) The efficiency of this operation can be improved by keeping intermediate values of `Num_Paths` in a two-dimensional array of `Positive` values. This keeps the function from having to recalculate values that it has already done. Design and code a version of `Num_Paths` that uses this approach.

(d) Show an invocation of the version of `Num_Paths` in part (c), including any array initialization necessary.

(e) How do the two versions of `Num_Paths` compare in terms of time efficiency? Space efficiency?

18. You are given the following function:

```
function Ulam (Num : in Integer) return Integer is
 Result : Integer;
begin
 if Num < 2 then
 Result := 1;
 elsif Num rem 2 = 0 then
 Result := Ulam (Num / 2);
 else
 Result := Ulam (3* Num + 1);
 end if;
 return Result;
end Ulam;
```

(a) What problems come up in verifying this function?

(b) How many recursive calls are made by the following initial calls:

```
Integer_IO.Put (Item => Ulam (7));
Integer_IO.Put (Item => Ulam (8));
Integer_IO.Put (Item => Ulam (15));
```

19. Explain the relationship between dynamic storage allocation and recursion.

20. What do we mean by *binding time*, and what does it have to do with recursion?

21. You are given the following values in `List`:

|     |     |     |     |     |     |     |     |     |      |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|
| 2   | 6   | 9   | 14  | 23  | 65  | 92  | 96  | 99  | 100  |
| (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) | (9) | (10) |

Show the contents of the run-time stack during the execution of this call to `Binary_Search`:

```
Binary_Search (List => List, Value => 99, Location => Location);
```

22. The input to the following two recursive routines is a pointer to a singly linked list of numbers, whose elements are unique (no duplicates) and unordered. Each node in the list contains two fields, `Info` (a number) and `Next` (a pointer to the next node).

(a) Write a recursive function, `Min_Loc`, that inputs a pointer to a list of unordered numbers and returns a pointer to the node that contains the minimum value in the list.

(b) Write a recursive procedure, `Sort`, that inputs a pointer to an unordered list of numbers and reorders the values in the list from smallest to largest. This procedure may call the recursive `Min_Loc` function that you wrote in part (a). (Hint: It is easier to swap the values in the `Info` part of the nodes than to try to reorder the nodes in the list.)

23. *True or False?* If you answer false, correct the statement.

A recursive solution should be used when

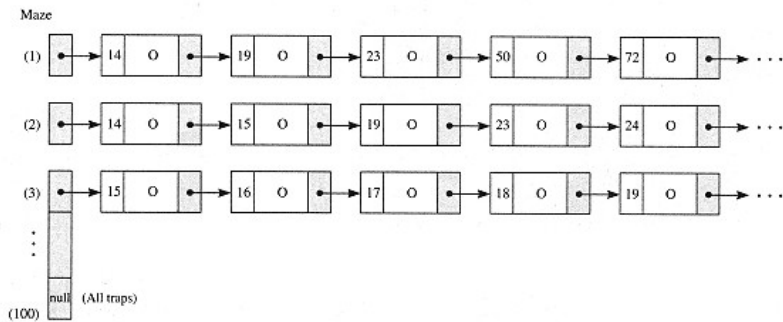
- (a) computing time is critical.
- (b) the nonrecursive solution would be longer and more difficult to write.
- (c) computing space is critical.
- (d) your instructor says to use recursion.

Exercises 24–27 refer to the Maze case study at the end of the chapter.

24. What are the base case(s) and general case(s) of the `Try_To_Escape` procedure?

25. Modify the maze program so that if an escape route is found, the maze with the route marked with \* symbols is displayed.

26. A very large maze (`Maze_Size = 100`) contains very few open paths relative to the number of trap positions. To save space, you decide to represent the Maze ADO as an array of `Maze_Size` pointers, where each pointer accesses a list of Open, Tried, or Finish path positions in one row, as illustrated below:



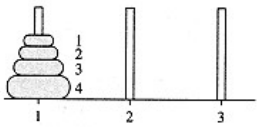
The numbers in these nodes are column numbers. From the drawing we can see that for row 1, columns 14, 19, 50, 72, ... are all open and that column 23 of this row contains the exit.

- Write type declarations to support `Maze_Type`.
- Describe algorithms for the revised `Mark` and `Symbol_At` operations.
- Describe how this change of data representation affects the rest of program `Escape`.

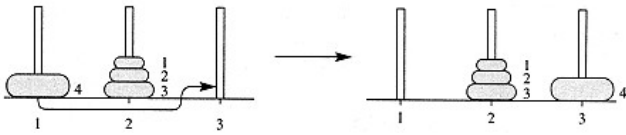
### Programming Problems

1. One of your first toys may have been three pegs with colored circles of different diameters. If so, you probably spent countless hours moving the circles from one peg to another. If we put some constraints on how the circles or disks can be moved, we have an adult game called the Towers of Hanoi. When the game begins, all the circles are on the first peg in order by size, with the smallest on the top. The object of the game is to move the circles, one at a time, to the third peg. The catch is that a circle cannot be placed on top of one that is smaller in diameter. The middle peg can be used as an auxiliary peg, but it must be empty at the beginning and the end of the game.

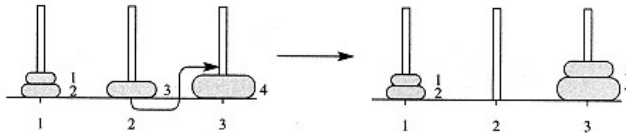
To get a feel for how this might be done, let's look at some sketches of what the configuration must be at certain points if a solution is possible. We use four circles or disks. The beginning configuration is



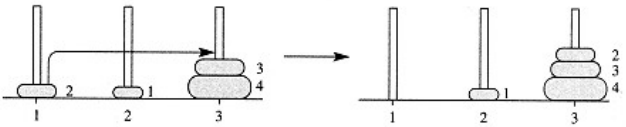
To move the largest circle (circle 4) to peg 3, we must move the three smaller circles to peg 2. Then circle 4 can be moved into its final place:



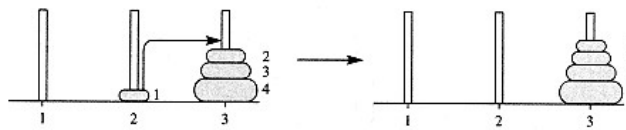
Let's assume that we can do this. Now, to move the next largest circle (circle 3) into place, we must move the two circles on top of it onto an auxiliary peg (peg 1 in this case):



To get circle 2 into place, we must move circle 1 to another peg, freeing circle 2 to be moved to its place on peg 3:



We can now move the last circle (circle 1) into its final place, finishing the game:



Notice that to free circle 4 we had to move three circles to another peg. To free circle 3, we had to move two circles to another peg. To free circle 2, we had to move one circle to another peg. This sounds like a recursive algorithm: to free the  $N$ th circle, we have to move  $N - 1$  circles.

Each stage can be thought of as beginning again with three pegs, but with one fewer circle each time. Let's see if we can summarize this process, using  $N$  instead of an actual number.

### To Get $N$ Circles Moved from Peg 1 to Peg 3

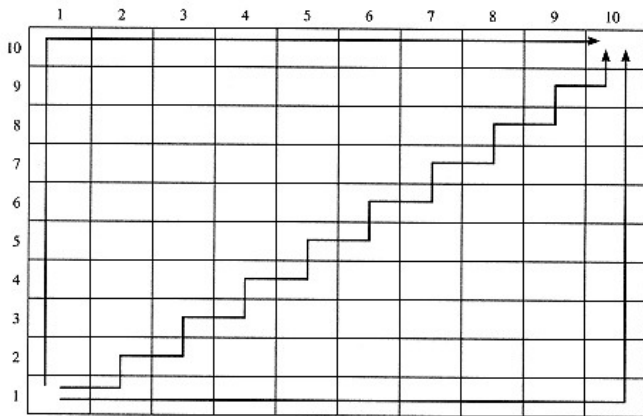
```
Get $N - 1$ circles moved from peg 1 to peg 2
Move Nth circle from peg 1 to peg 3
Get $N - 1$ circles moved from peg 2 to peg 3
```

This algorithm certainly sounds simple. You'd think surely there must be more to it, but this is really all that's necessary.

Your task is to write a program that contains a recursive procedure that implements this algorithm. You can't actually move disks, of course, but you can print out a message to do so. Your program should prompt the user to enter the number of disks that are on the first peg. It should then display a list of instructions that someone could follow to move all the disks to the third peg. Your output for three disks should look like this:

```
Move disk from peg 1 to peg 3
Move disk from peg 1 to peg 2
Move disk from peg 3 to peg 2
Move disk from peg 1 to peg 3
Move disk from peg 2 to peg 1
Move disk from peg 2 to peg 3
Move disk from peg 1 to peg 3
Game complete!
```

2. In this assignment you are to write and compare two procedures that count the number of paths possible to move in a two-dimensional square grid from row 1, column 1 to row  $N$ , column  $N$ . Steps are restricted to going up or to the right, but not diagonally. The illustration below shows three of many paths, if  $N = 10$ :





The functions that you are to write have the following specification:

```
function Num_Paths (Row : in Positive; -- Starting Row
 Col : in Positive; -- Starting Column
 N : in Positive) -- Maximum row/column
 return Positive; -- Number of paths possible
--
-- Returns the number of paths possible to move in a two-dimensional
-- square grid from starting Row/Col to row N/column N.
--
-- Preconditions: 1 <= Row <= N
-- 1 <= Col <= N
--
-- Postconditions: Num_Paths = number of paths possible to move
-- from starting row and column to terminal position
-- at row N/column N.
```

Both versions of Num\_Paths must be recursive functions.

### **Version 1**

The first version (call it Num\_Paths\_1) uses the "brute force" approach:

```
if Row = N or Col = N then
 Result := 1
else
 Result := sum of Num_Paths starting at next row/same
 column and Num_Paths starting at same
 row/next column.
```

After you have coded the function, walk through the execution of Num\_Paths (1, 1, 4) by hand. Why is this algorithm inefficient? (Save the walkthrough and the answer to the question to turn in with your program listings.)

### **Version 2**

The efficiency of this operation can be improved by keeping intermediate values of Num\_Paths in a two-dimensional array of Natural values. This keeps the function from having to recalculate values that it has already done. Design and code a version of Num\_Paths (call it Num\_Paths\_2) that uses this approach. Be careful about where and how the array is declared and initialized! (You may want to enclose this array in a package with suitable operations.)

### **Testing the Implementations**

Write a test driver to test these functions. The driver should prompt the tester to input the value of  $N$ , then call each version of Num\_Paths to calculate the number of paths. The result of each call should be printed. The initial (nonrecursive) calls are

```
Path_Count := Num_Paths_1 (1, 1, N);
```

and

```
Path_Count := Num_Paths_2 (1, 1, N);
```

The test driver should continue prompting the tester to supply values of  $N$  until 0 is input.

Run the program, using values of 1, 2, 3, . . ., and so on, for  $N$ . Both versions should get the same results. At what point does the answer get too large for type Positive? (You may want to change the result type of the functions to Float.)

### Comparing the Implementations

To compare the two versions of `Num_Paths`, add a statement to each function to increment a variable called `Call_Count`. `Call_Count` is used to see how many times the function is called. This variable should be declared as a global variable in the test driver. Before each invocation of a `Num_Paths` function, initialize `Call_Count` to 0. After each call, print out `Call_Count`. Record the results of `Call_Count` for each function for  $N$  values of 1 to 10. Answer the following questions:

- (1) How do the two versions of `Num_Paths` compare in terms of execution-time efficiency?
- (2) How do the two versions of `Num_Paths` compare in terms of space efficiency?

### Turn In

- (a) walkthrough of `Num_Paths_1` (1, 1, 4)(Why is it inefficient?)
- (b) program listing of test driver, including the two versions of `Num_Paths`
- (c) the answers to questions (1) and (2) in the previous section
- (d) the following chart filled in with values calculated by your program:

| $N$ | Number<br>of paths | Call_Count               |                          |
|-----|--------------------|--------------------------|--------------------------|
|     |                    | <code>Num_Paths_1</code> | <code>Num_Paths_2</code> |
| 1   |                    |                          |                          |
| 2   |                    |                          |                          |
| 3   |                    |                          |                          |
| 4   |                    |                          |                          |
| 5   |                    |                          |                          |
| 6   |                    |                          |                          |
| 7   |                    |                          |                          |
| 8   |                    |                          |                          |
| 9   |                    |                          |                          |
| 10  |                    |                          |                          |

3. Your assignment is to (1) write a procedure that computes an approximation to a definite integral that is within a given tolerance of the exact answer, (2) write a sorting procedure that helps monitor the behavior of the integration function, and (3) test your procedure on several prescribed integrals.

### Method

The integration procedure implements an adaptive algorithm based on the trapezoid rule of integration and its associated error estimate. The trapezoid rule for integration is

$$\int_a^b f(x)dx = (f(a) + f(b)) * (b - a)/2$$

If the interval  $[a, b]$  is divided into two equal subintervals, another estimate of the integral can be obtained by applying the trapezoid rule to each half interval and adding the two results. This new result is normally more accurate, and an estimate of the error in the better result is given by

$$\text{Error} = (T(a, b) - (T(a, c) + T(c, b))) / 3.0$$

where  $c = (a + b)/2$  and  $T(x, y)$  is the result of applying the trapezoid rule to  $f$  on the interval  $[x, y]$ .

The integral is to be computed to within a given degree of accuracy by a divide-and-conquer strategy. With the above formulas, the integral can be approximated and the error in the approximation can be estimated. If the absolute value of the error is small enough, the calculation terminates. Otherwise, the integral is computed on each half interval separately with an error tolerance that is half the original one.

All the  $x$  values at which the function is evaluated are to be stored in an array. After the integration routine terminates, these values should be sorted and printed out. (Use any sorting method to sort the array of values in increasing order). This helps monitor the behavior of the integration algorithm by showing where the function  $F$  was evaluated. The evaluations should be concentrated where the function is badly behaved (that is, poorly approximated by straight-line pieces).

### Specification

The specification of the integration procedure is

generic

```
type Real is digits <>;
type Array_Type is array (Positive range <>) of Real;
with function F(X : in Real) return Real;
```

```
procedure Integrate (A : in Real;
 B : in Real;
 Epsilon : in Real;
 Evaluations : out Array_Type;
 Last_Index : out Positive;
 Error : out Boolean;
 Result : out Real);
```

**Real** is the floating-point type you wish to use.

**Array\_Type** is the type for out parameter **Evaluations** (described below).

**F** is the function to be integrated.

**A** is the left endpoint of the interval.

**B** is the right endpoint of the interval.

**Epsilon** is the maximum allowed absolute value of the error.

**Evaluations** all of the  $x$  values at which the function  $F$  was evaluated. The procedure evaluates the function  $F$  a maximum of **Evaluations** ' **Last** times. **Evaluations** ' **First** must be 1.

**Last\_Index** the position of the last value in **Evaluations** (This is also the number of times the function  $F$  was evaluated).

**Error** is **True** if more than **Evaluations** ' **Last** function evaluations are needed to satisfy the error test.

**Result** the estimate of the integral.

### Implementation Details

1. Instantiate an integration procedure for each of the functions whose integrals are to be estimated. Use the predefined type **Float** for **Real**. Later you may wish to experiment with higher or lower precision types.

2. To prevent the possibility of infinite recursion, the integration procedure should terminate if the function `F` is called `Evaluations'Last` times. In this case, the best available approximation to the integral should be returned and an error flag should be set.
3. The primary cost of using an integration routine is in the cost of evaluating the function `F`. Write your code so that it never evaluates the function twice at the same point.
4. The divide-and-conquer approach should be implemented by a recursive procedure nested within `Integrate`. This recursive procedure isn't called directly, but is called by the nonrecursive driver procedure `Integrate`, described above.
5. The straightforward way to compute the midpoint of an interval is not the best. To ensure that the computed midpoint of an interval  $[a, b]$  is not outside the interval, use the formula  $a + (b - a)/2$ .
6. The main program should do the following, once for each integration procedure instantiated with the functions in the test data:
  - (a) Call each integration procedure with the appropriate arguments.
  - (b) Print the estimate of the integral.
  - (c) Print an error message if `Error` is true.
  - (d) Sort the array `Evaluations`.
  - (e) Print the sorted values.

### Testing the Function

Test your integration function by computing the following integrals.

| $F(x)$                    | Left | Right | Epsilon   | Evaluations'Last |
|---------------------------|------|-------|-----------|------------------|
| 1                         | 0.0  | 1.0   | $10^{-3}$ | 5                |
| $x$                       | -1.0 | 3.0   | $10^{-5}$ | 3                |
| $x$                       | -1.0 | 3.0   | $10^{-5}$ | 2                |
| $e^{-x^2}$                | 0.0  | 5.0   | $10^{-3}$ | 1000             |
| $(\frac{1}{3} - x)^{1/3}$ | 0.0  | 1.0   | $10^{-2}$ | 10               |
| $(\frac{1}{3} - x)^{1/3}$ | 0.0  | 1.0   | $10^{-3}$ | 1000             |
| $\frac{4}{(1 + x^2)}$     | 0.0  | 1.0   | $10^{-3}$ | 1000             |

### Notes

Ada 83 does not provide an operator for raising a Float value to a Float number power. However most Ada 83 compilers come with a math package containing a function that implements this operation. Most exponentiation functions do not allow you to raise a negative number to a float number power. This problem occurs with the fifth and sixth functions above. Because we are calculating a cube root, you can take the absolute value of the base, raise it to a power, and then negate if the original value was negative. If your math library does not include an exponential function, use a value of 2.718281828 for  $e$ .

*(This programming assignment was written by Alan Cline and David Scott.)*

4. Modify the Maze program to find *all* the paths from a given start point to a given finish point. At the end, display the maze with the shortest path marked with stones ('.').

**12—****Binary Search Trees****GOALS**

After reading this chapter, you should be able to

- define and use the following terminology:

binary tree

binary search tree

root

leaf

parent

child

ancestor

descendant

level

height

subtree

- define a binary search tree at the logical level
- show what a binary search tree would look like after a series of insertions and deletions

- implement the following binary search tree algorithms in Ada:

creating an empty tree

inserting an element

deleting an element

retrieving an element

modifying an element

counting the number of nodes

traversing a tree in preorder, inorder, and postorder

We have discussed some of the advantages of using a linear linked list to store value-ordered information. One of the drawbacks of using a linear linked list is the time it takes to search a long list. A sequential search of (possibly) all the nodes in the whole list is an  $O(N)$  operation. In Chapter 2 we saw how a binary search could find an element in an ordered list stored sequentially in an array. The binary search is an  $O(\log_2 N)$  operation. It would be nice if we could binary search a linked list, but there is no practical way to find the midpoint of a linked list of nodes. We can, however, reorganize the list's elements into a linked structure that is just perfect for binary searching: the *binary search tree*. The binary search tree provides us with a structure that retains the flexibility of a linked list while allowing quicker  $O(\log_2 N)$  access to any node in the list.

This chapter introduces some basic tree vocabulary and then develops the algorithms and implementations of the operations needed to use a binary search tree. As with previous data structures, we use a generic package to implement binary search trees. The user who instantiates the package supplies the element type and a function that returns the *unique* key of that element. (In the figures that illustrate trees, we only show the key values for simplicity.)

We assume that each element stored in the binary search tree is a record that contains a number of fields, including a field that is designated as a unique key.

## The Abstract Level

---

### Binary Tree

A structure with a unique starting node (the root), in which each node is capable of having two child nodes, and in which a unique path exists from the root to every other node.

### Child

The successor of a node in a binary tree. Each node may have 0, 1, or 2 children.

### Root

The top, or base, node of a tree structure.

### Parent

The predecessor of a node in a binary tree. Each node except the root node has one parent.

---

Each node in a singly linked list may point to one other node: the one whose value follows it. Thus a singly linked list is a *linear* structure; each node in the list has a unique successor. A **binary tree** is a structure in which each node is capable of having *two* successor nodes, called **children**. Each of the children, being nodes in the binary tree, can also have up to two child nodes, and these children can also have up to two children, and so on, giving the tree its branching structure. The "beginning" of the tree is a unique starting node called the **root**. Each node in the binary tree, except for the root node has one predecessor—its **parent**.

Figure 12-1 depicts a binary tree. The root node of this binary tree contains the value A. Each node in the tree may have 0, 1, or 2 children. The node to the left of a node, if it exists, is called its *left child*. For instance, the left child of the root node in Figure 12-1 contains the value B. The node to the right of a node, if it exists, is its *right child*. The right

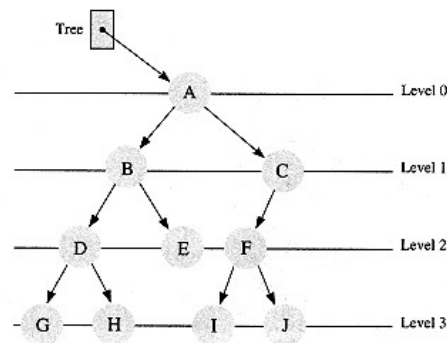
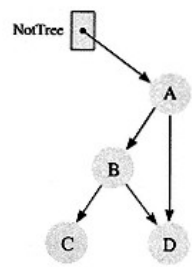


Figure 12-1  
A Binary Tree

|                                 |
|---------------------------------|
| <b>Leaf Node</b>                |
| Tree node that has no children. |

child of the root node in Figure 12-1 contains the value C. The root node is the parent of the nodes containing B and C. If a node in the tree has no children, it is called a **leaf**. For instance, the nodes containing G, H, E, I, and J are leaf nodes.

In addition to specifying that a node may have up to two children, the definition of a binary tree states that a unique path exists from the root to every other node. This means that every node (except the root) has a unique parent. In the structure pictured below, the nodes have the correct number of children, but the unique path rule is violated: There are two paths from the root to the node containing D. Therefore this structure is not a binary tree:



|                                                                  |
|------------------------------------------------------------------|
| <b>Subtree</b>                                                   |
| A portion of a tree. Each node in a tree is a root of a subtree. |

|                                                                                                      |
|------------------------------------------------------------------------------------------------------|
| <b>Descendant</b>                                                                                    |
| A node in a tree that is the child of another node, or the child of another descendant of that node. |

|                                                                                                      |
|------------------------------------------------------------------------------------------------------|
| <b>Ancestor</b>                                                                                      |
| A node in a tree that is the parent of another node, or the parent of another ancestor of that node. |

Note, in Figure 12-1, that each of the root node's children is itself the root of a smaller binary tree, or **subtree**. The root node's left child, containing B, is the root of its *left subtree*, whereas the right child, containing C, is the root of its *right subtree*. In fact, any node in the tree can be considered the root node of a subtree. The subtree whose root node has the value B also includes the nodes with values D, G, H, and E. These nodes are the **descendants** of the node containing B. The descendants of the node containing C are the nodes with the values F, I, and J. A node is the **ancestor** of another node if it is the parent of the node, or the parent of some other ancestor of that node. (Yes, this is a recursive definition.) The ancestors of the node with the value G are the nodes containing D, B, and A. Obviously, the root of the tree is the ancestor of every other node in the tree.

|                                            |
|--------------------------------------------|
| <b>Level</b>                               |
| The distance of a node from the root node. |

The **level** of a node refers to its distance from the root. If we designate the level of the root as 0 (zero), the nodes in Figure 12-1 containing B and C are Level 1 nodes, the nodes containing D, E, and F are Level 2 nodes, and the nodes containing G, H, I, and J are Level 3 nodes.

|                                 |
|---------------------------------|
| <b>Height</b>                   |
| The number of levels in a tree. |

The number of levels in a tree determines its **height**. The maximum number of nodes *at any level* is  $2^L$  (where  $L$  is the level number). However, often levels do not contain the maximum number of nodes. For instance, in Figure 12-1, Level 2 could contain four nodes, but because the node containing C in Level 1 only has one child, Level 2 contains three nodes. Level 3, which could contain eight nodes, only has four. We could make many differently shaped binary trees out of the 10 nodes in this tree. Figure 12-2 shows three more arrangements of these 10 nodes.

It is easy to see that the *maximum number of levels* in a binary tree with  $N$  nodes is  $N$  [Figure 12-2(c)]. What is the minimum number of levels? If we fill the tree by giving every node in each level two children until we run out of nodes, the tree has  $\lceil 7.05 \rceil$  are both 8. The following table shows the minimum tree height possible for various numbers of nodes.

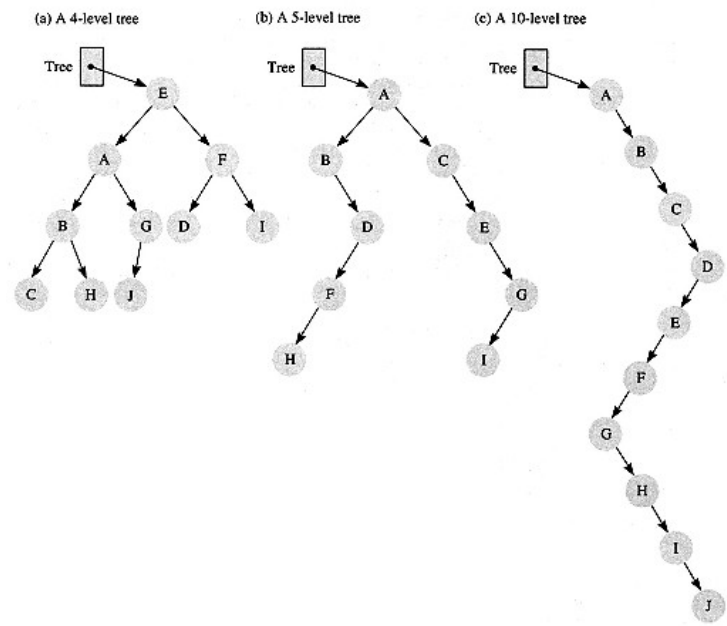


Figure 12-2  
Binary Trees with 10 Nodes

| Minimum Heights of Binary Trees |                 |                                                        |
|---------------------------------|-----------------|--------------------------------------------------------|
| $N$<br>(Number of Nodes)        | $\log_2(N + 1)$ | $\lceil \log_2(N + 1) \rceil$<br>(Minimum Tree Height) |
| 1                               | 1.000           | 1                                                      |
| 2                               | 1.585           | 2                                                      |
| 3                               | 2.000           | 2                                                      |
| 4                               | 2.322           | 3                                                      |
| 5                               | 2.585           | 3                                                      |
| 6                               | 2.807           | 3                                                      |
| 7                               | 3.000           | 3                                                      |
| 8                               | 3.170           | 4                                                      |
| 9                               | 3.322           | 4                                                      |
| 10                              | 3.459           | 4                                                      |
| 100                             | 6.658           | 7                                                      |
| 1,000                           | 9.967           | 10                                                     |
| 10,000                          | 13.288          | 14                                                     |
| 100,000                         | 16.610          | 17                                                     |
| 1,000,000                       | 19.932          | 20                                                     |



The height of a tree is the critical factor in determining how efficiently we can search for elements. Consider the maximum-height tree in Figure 12-2(c). If we begin searching at the root node and follow the pointers from one node to the next, accessing the node with the value J (the farthest from the root) is an  $O(N)$  operation—no better than searching a linear list! On the other hand, given the minimum-height tree depicted in Figure 12-2(a), to access the node containing J, we only have to look at three other nodes—the ones containing E, A, and G—before we find J. Thus, if the tree is of minimum height, its *structure* supports  $O(\log_2 N)$  access to any element.

However, the *arrangement* of the values in the tree pictured in Figure 12-2(a) does not lend itself to quick searching. Let's say that we want to find the value G. We begin searching at the root of the tree. This node contains E, not G, so we need to keep searching. But which of its children should we look at next, the right or the left? There is no special order to the nodes, so we have to check *both* subtrees. We could search the tree, level by level, until we come across the value we are searching for. But that's an  $O(N)$  search operation, which is no better than searching a linked list!

To support  $O(\log_2 N)$  searching, we add a special property to the binary tree structure. We put all the nodes with values *smaller* than the value in the root into its left subtree, and all the nodes with values *larger* than the value in the root into its right subtree. Figure 12-3 shows the nodes from Figure 12-2(a) rearranged to satisfy this property. The root node, which contains E, has two subtrees. The left subtree contains all the values smaller than E and the right subtree contains all the values larger than E.

Searching for the value G, we look first in the root node. G is larger than E, so we know that G must be in the root node's right subtree. The right child of the root node contains H. Now what? Do we go to the right or to the left? This subtree is also arranged according to the binary search property: The nodes with smaller values are to the left, and the nodes with larger values are to the right. The value of this node, H, is greater than G, so

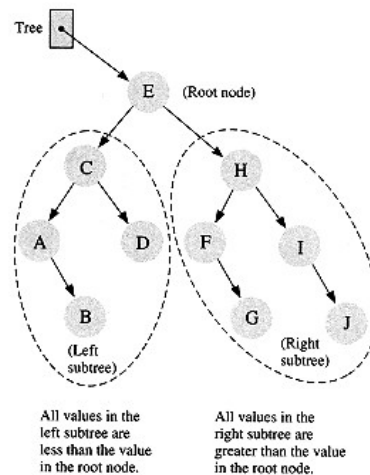


Figure 12-3  
A Binary Search Tree

we search to its left. The left child of this node contains the value F, which is smaller than G, so we reapply the rule and move to the right. The node to the right contains G; we have found the node we were searching for.

---

#### Binary Search Tree

A binary tree in which the key value in every node is greater than the key values in its left subtree and less than the key values in its right subtree.

---

A binary tree with this special property is called a **binary search tree**. Like any binary tree, it gets its branching structure by allowing each node to have up to two child nodes. It gets its easy-to-search structure by maintaining the binary search property: The left child of any node (if there is one) is the root of the subtree that contains all of the values smaller than the node. The right child of any node (if there is one) is the root of the subtree that contains all of the values that are larger than the node.

Four comparisons instead of up to 10 doesn't sound like such a big deal, but as the number of elements in the structure increases, the difference becomes impressive. In the worst case—searching for the last node in a linear linked list—you must look at every node in the list; on the average you must search half the list. If the list contains one million nodes, you must make one million comparisons to find the last node! If the one million nodes were arranged in a binary search tree of minimum height, you would never make more than 20 comparisons, no matter which node you were seeking!

#### *A Binary Search Tree Package Declaration*

Now let's specify the logical structure and the operations defined for a Binary Search Tree ADT. There are many operations that one might want to define to manipulate a binary search tree. We have included a "basic" set of operations in Specification 12-1; some other operations are specified in the exercises at the end of the chapter. The operations are similar to those in our sorted list class. From the application programmer's point of view, the important thing is to be able to access and manipulate the data stored in the tree. Of course, the algorithms we develop to implement the operations are different from those for the list operations.

#### Specification 12-1— Binary Search Tree

generic

```

type Element_Type is private; -- The type of element in the list
type Key_Type is limited private; -- The type of key in the element

-- The user must supply a function that returns the Key of an Element.
with function Key_Of (Element : in Element_Type) return Key_Type;
-- The user must supply functions for comparing Keys.
with function "=" (Left : in Key_Type; Right : in Key_Type) return Boolean;
with function "<" (Left : in Key_Type; Right : in Key_Type) return Boolean;
```

package Binary\_Search\_Tree is

-- This package implements a binary search tree.

-- Each element in the tree has a unique key.

```

-- The elements in the tree are ordered from smallest to largest using the "<"
-- function supplied as a generic parameter. The placement of each element into
-- the binary tree must satisfy the binary search property: The value of the key
-- of an element is greater than the value of the key of any element in its left
-- subtree, and less than the value of the key of any element in its right subtree.

type Tree_Type is limited private;

type Traversal_Order is (Inorder, Preorder, Postorder);

DUPLICATE_KEY : exception;
KEY_ERROR : exception;
OVERFLOW : exception;

-- Constructors

procedure Clear (Tree : in out Tree_Type);
-- This procedure removes all elements from Tree.

-- Preconditions : None

-- Postconditions : Tree is empty.

procedure Insert (Tree : in out Tree_Type;
 Item : in Element_Type);
-- This procedure adds Item to Tree.

-- Preconditions : None

-- Postconditions : Tree = original Tree + Item

-- Exceptions : OVERFLOW If there is no room for Item. Tree is unchanged.
-- DUPLICATE_KEY If an element already exists in the tree
-- with the same key as Item. Tree is unchanged.

procedure Delete (Tree : in out Tree_Type;
 Key : in Key_Type);
-- Deletes the element containing Key from Tree.

-- Preconditions : None

-- Postconditions : Tree = original Tree with Key removed

-- Exceptions : KEY_ERROR If there is no element with Key in Tree.
-- Tree is unchanged.

procedure Modify (Tree : in out Tree_Type;
 Element : in Element_Type);
-- Replace existing Tree element with same key as Element.

```

```

-- Preconditions : None

-- Postconditions : Tree = original Tree with value of Element replacing
-- like-keyed original element.

-- Exceptions : KEY_ERROR If there is no element with the key of
-- Element in Tree. Tree is unchanged.

-- Observers

procedure Retrieve (Tree : in Tree_Type;
 Key : in Key_Type;
 Element : out Element_Type);
-- A copy of the element that contains Key is returned.

-- Preconditions : None

-- Postconditions : Element = copy of element containing Key
-- Tree is unchanged.

-- Exceptions : KEY_ERROR If there is no element with the
-- Key in Tree. Tree is unchanged.

function Full (Tree : in Tree_Type) return Boolean;
-- Determines whether Tree is full.

-- Preconditions : None

-- Postconditions : Full = (Tree is full)

function Empty (Tree : in Tree_Type) return Boolean;
-- Determines whether Tree is empty.

-- Preconditions : None

-- Postconditions : Empty = (Tree is empty)

function Size (Tree : in Tree_Type) return Natural;
-- Determines the number of elements in Tree.

-- Preconditions : None

-- Postconditions : Size = number of elements in Tree

-- Iterator

generic
 -- The user must supply a procedure that will be called once for every
 -- element in the Tree. Process may not change the key of Element.
 with procedure Process (Element : in out Element_Type);

```

```

procedure Traverse (Tree : in out Tree_Type;
 Order : in Traversal_Order);
-- Calls procedure Process once for every element in the Tree.
-- The order in which nodes are "processed" is specified by parameter Order.

-- Preconditions : None

-- Postconditions : Every element in Tree is passed to a call of Procedure Process.

-- Exceptions : KEY_ERROR If Process attempts to change the key of any element.

private

-- We fill this in later.

end Binary_Search_Tree;

```

### Binary Tree Traversals

As with the List, to traverse a tree means to process all its nodes—for example, to display *all* of the values in the tree or to delete *all* the elements in the tree. What orders are possible? For linear lists, the most common orders are forward (from smallest to largest) and backward (largest to smallest). Other orders are also possible for linear lists. For example, we could process the odd elements (the first, third, fifth, and so on) and then process the even elements. Such a processing order is rare. Lists are almost always traversed forward or backward. There are three common orders for traversing binary trees: *Inorder*, *Preorder*, and *Postorder*.

### Inorder Traversals

The definition of an Inorder traversal of a binary tree is recursive:

#### Inorder

```

if the tree is not empty
 traverse the left subtree in Inorder
 process the root
 traverse the right subtree in Inorder
end if

```

Notice that an Inorder traversal processes each node *in between* its left and right subtrees—that's how it gets its name. This processing is illustrated in Figure 12-4. When a *binary search* tree is traversed Inorder, the nodes are processed in ascending order by key. An Inorder traversal of a binary search tree is therefore similar to the forward traversal of a sorted list.

Let's look at the recursive Inorder traversal definition as an algorithm. Figure 12-5 shows a binary search tree containing 11 nodes. We start with the root node, P. The algorithm first says to traverse P's left subtree in Inorder. The root node of this subtree is F. The algorithm says to first traverse F's left subtree in Inorder. The root node of this subtree is B. The algorithm first says to traverse B's left subtree in Inorder. Now B's left subtree is empty. Traversing an empty tree in Inorder is easy—we do nothing. Now that we have

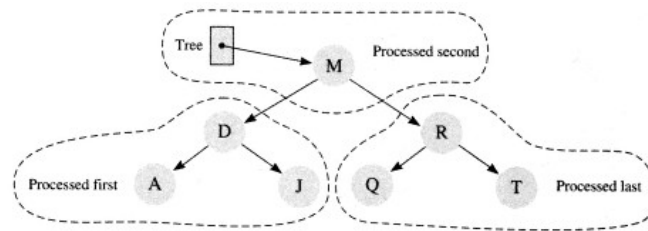


Figure 12-4  
Processing All the Nodes Inorder

completed traversing B's left subtree, the second step of the algorithm says to process the root of the subtree: the node containing B. The third step is to traverse B's right subtree in Inorder. Again this is an easy task as the subtree is empty. Where are we now? We have just completed traversing F's left subtree in Inorder. So now we carry out the second step of our algorithm and process the root of the subtree: the node containing F. The third step tells us to traverse F's right subtree in Inorder. Can you determine in what order the nodes in F's right subtree are processed? How about for the rest of the tree? The order of Inorder processing of the entire tree illustrated in Figure 12-5 is

B F G H P R S T W Y Z

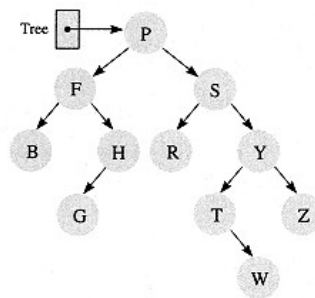


Figure 12-5  
Three Tree Traversals

### Preorder Traversal

The definition for a Preorder traversal of a binary tree is

#### Preorder

```

if the tree is not empty
 process the root
 traverse the left subtree in Preorder
 traverse the right subtree in Preorder
end if

```

How is this definition different from that of the Inorder traversal? The order of the steps has changed. Here we process the root *before* we traverse its two subtrees. Given the tree in Figure 12-5, a Preorder traversal would process the nodes in the order:

P F B H G S R Y T W Z

### Postorder Traversal

The definition for a Postorder traversal of a binary tree is

#### Postorder

```

if the tree is not empty
 traverse the left subtree in Postorder
 traverse the right subtree in Postorder
 process the root
end if

```

The order of the steps has changed again. Here we process the root *after* we traverse its two subtrees. Given the tree in Figure 12-5, a Postorder traversal would process the nodes in the order:

B G H F R W T Z Y S P

### The Application Level: Medical Emergency Response

You have just taken a job with a company that responds to potential medical emergencies reported by its clients. Each client is supplied with a small radio transmitter on a necklace or bracelet and a receiver that is connected to the telephone. To summon assistance, a client presses a button on the transmitter, which sends a signal to the receiver, which dials the number of the medical response company that has employed you. Here, the phone number of the caller is displayed on a small screen in front of an operator. Using this phone number, the operator looks up the name and address of the client and calls the appropriate authority to handle the emergency.

Another service offered by the company is called "Check In." At regular intervals, an operator phones a client to check on his or her status. Each client subscribing to this service selects his or her own time interval.

Currently all the information for clients is maintained on index cards. However, as the company is growing, searching the cards for the necessary information when a call is received is becoming dangerously slow. Your job is to "computerize" the stack of index cards. The program must, as a minimum, duplicate all of the services provided by the index card system. This includes finding the necessary information for a client (when given the phone number), determining what clients are due for a check-in call, adding and deleting clients, and changing client information (the most common changes are to intervals between check-ins). The program must also include a means for backing up all client information to disk and for restoring that information.

After talking with the company owner and all of the operators, you devise the following specification for your program:

**Specification:**  
**Medical Response**

***Function***

This program maintains information on the clients of a medical response company.

***Input***

A command specifies an operation to be done. Depending on the command, there may be additional input values (the data). Here are the commands and other data required:

| <i>Command</i> | <i>Data Required</i>                                                                                                                                                                            |
|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Find           | Phone Number                                                                                                                                                                                    |
| Add            | First Name<br>Last Name<br>Birthday (month, day, and year)<br>Phone Number<br>Street Address<br>City<br>State<br>Zip Code<br>Local Emergency Phone Number<br>Interval Between Check Ins (Hours) |
| Delete         | Phone Number                                                                                                                                                                                    |
| Interval       | Phone Number<br>New Interval                                                                                                                                                                    |
| Change         | First Name<br>Last Name<br>Birthday (month, day, and year)<br>Phone Number<br>Street Address<br>City<br>State<br>Zip Code<br>Local Emergency Phone Number<br>Interval Between Check Ins (Hours) |
| Backup         |                                                                                                                                                                                                 |
| Restore        |                                                                                                                                                                                                 |
| Quit           |                                                                                                                                                                                                 |

***Notes on Data Values***

1. No first name, last name, street address, or city contains more than 20 characters.
2. The state is a standard U.S. Postal Service two-character abbreviation.
3. The zip code consists of five digits.
4. A phone number consists of a three-digit area code and a seven-digit number.



**Output**

The output depends on the command. In the following table we do not include prompt messages or error messages:

| <i>Command</i> | <i>Output</i>                                                                   |
|----------------|---------------------------------------------------------------------------------|
| Find           | All client information                                                          |
| Check          | Client names and phone numbers for all clients whose check-in times have passed |
| Add            | Confirm addition                                                                |
| Delete         | Confirm deletion                                                                |
| Interval       | Confirm change of check-in interval                                             |
| Change         | Confirm change of data                                                          |
| Backup         | Number of clients                                                               |
| Restore        | Number of clients                                                               |
| Quit           |                                                                                 |

**Assumptions**

1. There are no duplicate phone numbers.
2. There are no duplicate names.

**The Design**

This problem is very similar to the automated address book we saw in Chapter 8. The main data object here is our *list of clients*. Let's call this object `Clients`. Because we need only one list of clients, this object can be implemented as an ADO. The operations on `Clients` are the commands given in our problem specification.

A client object is very similar to a friend object that we used in the automated address book. However, in order to implement the check-in feature, we need two more pieces of information: the interval between check-ins (given in hours) and the time of the last check-in. Because not all clients subscribe to the check-in service, we need some way of indicating this. We can use a check-in interval of zero for the clients that do not subscribe to the check-in service. Specification 12-2 gives our operations and the supporting types.

**Specification 12-2—  
Client ADO**

```
with Bounded_Strings;
with Calendar;
package Clients is

-- This abstract data object models a list of clients.

-- Bounded String Class
package String20 is new Bounded_Strings (Max => 20);
```

```

-- Bounded String Types
subtype Name_String is String20.Bounded_String;
subtype Address_String is String20.Bounded_String;
-- Fixed Length String Types
subtype State_String is String (1..2);
subtype Zip_String is String (1..5);
subtype Area_Code_String is String (1..3);
subtype Phone_Num_String is String (1..7);

subtype Hours_Type is Natural; -- For check-in time interval

type Name_Rec is -- Name class
 record
 First : Name_String;
 Last : Name_String;
 end record;

type Address_Rec is -- Address class
 record
 Street : Address_String;
 City : Address_String;
 State : State_String;
 Zip : Zip_String;
 end record;

type Phone_Rec is -- Phone number class
 record
 Area : Area_Code_String;
 Number : Phone_Num_String;
 end record;

type Client_Rec is -- Client class
 record
 Name : Name_Rec;
 Address : Address_Rec;
 Phone : Phone_Rec;
 Birthday : Calendar.Time;
 Emergency_Number : Phone_Rec;
 Interval : Hours_Type := 0;
 Last_Check : Calendar.Time := Calendar.Clock;
 end record;

-- Exceptions
DUPLICATE_PHONE : Exception;
MISSING_PHONE : Exception;

procedure Find (Phone : in Phone_Rec;
 Client : out Client_Rec);
-- This procedure looks up the Phone number in the client tree and
-- returns the information associated with that client.

-- Preconditions : None

```

```

-- Postconditions : Client = record with the given Phone.
-- The client tree is not changed.

-- Exceptions : MISSING_PHONE If the Phone is not found in the client tree.
-- The client tree is not changed.

procedure Check;
-- This procedure displays in alphabetical order (to the standard output file)
-- the names and phone numbers of all clients whose check-in time has passed.

-- Preconditions : None

-- Postconditions : The names and phone number of all clients whose check-in time
-- has passed are displayed.
-- The Last_Check time of each client displayed is set to the
-- current time.

procedure Interval (Phone : in Phone_Rec;
 New_Interval : in Hours_Type);
-- This procedure changes the check-in interval for the client with the given Phone.

-- Preconditions : None

-- Postconditions : The check-in interval for the client with the given Phone is
-- set to New_Interval.

-- Exceptions : MISSING_PHONE If the Phone is not found in the client tree.
-- The client tree is not changed.

procedure Add (Client : in Client_Rec);
-- This procedure adds a client to the client tree.

-- Preconditions : None

-- Postconditions : Client tree = Original Client tree + Client

-- Exceptions : DUPLICATE_PHONE If the Phone of client is already in the client
-- tree. The client tree is not changed.

procedure Delete (Phone : in Phone_Rec);
-- This procedure deletes a client from the client tree.
-- Phone identifies the client to delete.

-- Preconditions : None

-- Postconditions : Client tree = Original Client tree - Client

-- Exceptions : MISSING_PHONE If the Phone is not found in the
-- client tree. The client tree is
-- not changed.

```

```

procedure Change (Client : in Client_Rec);
-- This procedure replaces information on a client with new information.

-- Preconditions : None

-- Postconditions : The information in Client replaces the information for the client
-- in the client tree.

-- Exceptions : MISSING_PHONE If the name given in Client is not found in the
-- client tree. The client tree is not changed.

procedure Backup;
-- This procedure saves the current client tree to disk.

-- Preconditions : The disk is not full.

-- Postconditions : The current client tree is saved to disk for the next program run.

procedure Restore;
-- This procedure restores the client tree with the information stored on disk.

-- Preconditions : The client tree was previously backed up to disk.

-- Postconditions : The current client tree is replaced by information saved
-- previously on disk.

end Clients;

```

### ***The Implementation***

The body for the client ADO could easily be implemented with the sorted list package we developed in Chapter 8. However, we choose to use a binary search tree so that the Find operation can be  $O(\log_2 N)$  rather than  $O(N)$ . After all, when a potential medical emergency is reported, the operator must respond as quickly as possible. Our tree elements are client records keyed by phone numbers.

Most of the operations in this ADO are similar to those in the address book ADO we developed in Chapter 8. Procedure `Check` is more complicated than the traversals we instantiated for the address book. In particular, the specification requires us to display the clients in alphabetical order. If we traverse our tree in Inorder, we process the clients in order by phone number. Traversing in Preorder or Postorder also does not process the clients by name. Somehow we need to create a list of clients sorted by name rather than phone number. Such a list can be created by instantiating a second binary search tree package; this time the elements are ordered by name. The following algorithm uses these two trees to display an alphabetical listing of clients whose check-in times have passed:

## Check

1. Traverse the Phone ordered tree in any order.  
The Process checks to see if the check-in time for a client has passed.  
If it has passed, a copy of the client record is inserted into a name ordered tree.
2. Traverse the Name ordered tree in Inorder.  
The Process displays the name and phone number for a client.
3. Clear the Name ordered tree.

Body 12-1 implements our Clients specification. The first part of this body contains the specifications of subprograms required for our two tree package instantiations. Notice that we don't need to declare a function to compare two phone number records for equality; the predefined equality operator for records does just what we need. The bodies for these subprograms and some others used by the external client operations follow the instantiations. Finally come the bodies of all of Clients' operations.

### Body 12-1—

#### Client ADO Implements Specification 12-2

```
with Binary_Search_Tree;
with Sequential_IO;
with Text_IO;
package body Clients is

 -- Declare functions needed for instantiating two binary search tree packages
 -- For Phone ordered tree
 function Phone_Of (Client : in Client_Rec) return Phone_Rec;
 function "<" (Left : in Phone_Rec; Right : in Phone_Rec) return Boolean;
 -- For Name ordered tree
 function Name_Of (Client: in Client_Rec) return Name_Rec;
 function Equal (Left : in Name_Rec; Right : in Name_Rec) return Boolean;
 function "<" (Left : in Name_Rec; Right : in Name_Rec) return Boolean;

 -- Binary search tree with Phone number as key
 package Phone_Tree is new Binary_Search_Tree (Element_Type => Client_Rec,
 Key_Type => Phone_Rec,
 Key_Of => Phone_Of,
 "=" => "=",
 "<" => "<");

 -- Binary search tree with Name as key
 package Name_Tree is new Binary_Search_Tree (Element_Type => Client_Rec,
 Key_Type => Name_Rec,
 Key_Of => Name_Of,
 "=" => Equal,
 "<" => "<");

 -- Sequential file of Client Records
 package Client_IO is new Sequential_IO (Element_Type => Client_Rec);
```

```

-- The Client Object (a binary search tree) is used globally
Client_Tree : Phone_Tree.Tree_Type;
-- The sequential file for client backup is used globally
File_Name : constant String := "Client.Bin";
Backup_File : Client_IO.File_Type;

-- Local Subprogram Bodies

function Phone_Of (Client: in Client_Rec) return Phone_Rec is
begin
 return Client.Phone;
end Phone_Of;

function "<" (Left : in Phone_Rec; Right : in Phone_Rec) return Boolean is
 use String20;
begin
 return Left.Area & Left.Number <
 Right.Area & Right.Number;
end "<";

function Name_Of (Client: in Client_Rec) return Name_Rec is
begin
 return Client.Name;
end Name_Of;

function Equal (Left : in Name_Rec; Right : in Name_Rec) return Boolean is
begin
 return String20.Equal (Left.First, Right.First) and
 String20.Equal (Left.Last, Right.Last);
end Equal;

function "<" (Left : in Name_Rec; Right : in Name_Rec) return Boolean is
 use String20;
-- A name is "less than" another name if it comes before it alphabetically.
begin
 return Left.Last < Right.Last -- Check last names
 or
 (String20.Equal (Left.Last, Right.Last) and -- Only check first
 Left.First < Right.First); -- names if last names
end "<"; -- are equal.

procedure Put (Name : in Name_Rec) is
-- Puts the Name to the current output file
begin
 String20.Put (Item => Name.Last);
 Text_IO.Put (" ", " ");
 String20.Put (Item => Name.First);
end Put;

```

```

procedure Put (Phone : in Phone_Rec) is
-- Puts the Phone to the standard output file
begin
 Text_IO.Put ('(' & Phone.Area & ") " &
 Phone.Number(1..3) & '-' & Phone.Number(4..7));
end Put;

function "+" (Time : in Calendar.Time; Hours : in Hours_Type) return Calendar.Time is
-- This function adds a number of hours to a Time.
 use Calendar;
 Seconds_Per_Hour : constant := 3600;
begin
 return Time + Duration (Hours * Seconds_Per_Hour);
end "+";

-- External operations

procedure Find (Phone : in Phone_Rec;
 Client : out Client_Rec) is
begin
 Phone_Tree.Retrieve (Tree => Client_Tree,
 Key => Phone,
 Element => Client);
exception
 when Phone_Tree.KEY_ERROR =>
 raise MISSING_PHONE; -- Convert Tree exception to client tree exception
end Find;

procedure Check is

Current_Time : constant Calendar.Time := Calendar.Clock;
-- A local tree for displaying information in order by name
Alpha_Tree : Name_Tree.Tree_Type;

-- "Process" procedure for traversing Alpha_Tree (ordered by Name)
procedure Display_One_Client (Client : in out Client_Rec) is
-- Displays the Name and Phone Number of one client
begin
 Put (Client.Name);
 Text_IO.Put (" ");
 Put (Client.Phone);
 Text_IO.New_Line;
end Display_One_Client;

-- "Process" procedure for traversing Client_Tree (ordered by Phone)
procedure Check_One_Client (Client : in out Client_Rec) is
-- Checks to see if the client's Check-In time has passed. If it has, the
-- client is inserted into the alphabetic client tree and their record is
-- updated to reflect the next Check-In time.
 use Calendar;

```

```

begin -- Check_One_Client
 -- Do we need to check on this client?
 if Client.Interval > 0 and then
 Current_Time > Client.Last_Check + Client.Interval then
 -- Update the last Check-In time
 Client.Last_Check := Current_Time;
 -- Add the client to the alphabetical tree
 Name_Tree.Insert (Tree => Alpha_Tree, Item => Client);
 end if;
end Check_One_Client;

-- Instantiate traversal procedures
procedure Display_All is new Name_Tree.Traverse (Process => Display_One_Client);
procedure Check_All is new Phone_Tree.Traverse (Process => Check_One_Client);

begin -- Check
 -- Find all Clients who need to be checked
 Check_All (Tree => Client_Tree, Order => Phone_Tree.Inorder);
 -- Display all Clients found
 Display_All (Tree => Alpha_Tree, Order => Name_Tree.Inorder);
 Name_Tree.Clear (Alpha_Tree);
end Check;

procedure Interval (Phone : in Phone_Rec;
 New_Interval : in Hours_Type) is
 Client : Client_Rec;
begin
 -- Get current Client information
 Phone_Tree.Retrieve (Tree => Client_Tree,
 Key => Phone,
 Element => Client);
 -- Change the interval
 Client.Interval := New_Interval;
 -- Update the tree
 Phone_Tree.Modify (Tree => Client_Tree,
 Element => Client);
exception
 when Phone_Tree.KEY_ERROR =>
 raise MISSING_PHONE; -- Convert tree exception to client tree exception
end Interval;

procedure Add (Client : in Client_Rec) is
begin
 Phone_Tree.Insert (Tree => Client_Tree, Item => Client);
exception
 when Phone_Tree.DUPLICATE_KEY =>
 raise DUPLICATE_PHONE; -- Convert tree exception to client tree exception
end Add;

```



```

procedure Delete (Phone: in Phone_Rec) is
begin
 Phone_Tree.Delete Tree => Client_Tree, Key => Phone);
exception
 when Phone_Tree.KEY_ERROR =>
 raise MISSING_PHONE; -- Convert tree exception to client tree exception
end Delete;

```

```

procedure Change (Client : in Client_Rec) is
begin
 Phone_Tree.Modify Tree => Client_Tree,
 Element => Client);
exception
 when Phone_Tree.KEY_ERROR =>
 raise MISSING_PHONE; -- Convert tree exception to client tree exception
end Change;

```

```

-- "Process" for the backup traverse procedure
procedure Write (Client : in out Client_Rec) is
begin
 Client_IO.Write (File => Backup_File, Item => Client);
end Write;

-- Instantiate the backup traverse procedure
procedure Write_All is new Phone_Tree.Traverse (Process => Write);

```

```

procedure Backup is
begin
 Client_IO.Create (File => Backup_File, Name => File_Name);
 -- Call the backup traverse procedure. Write out tree in Preorder
 -- so that the same shape tree will be produced by Restore
 Write_All (Tree => Client_Tree, Order => Phone_Tree.Preorder);
 Client_IO.Close (Backup_File);
end Backup;

```

```

procedure Restore is
 Client : Client_Rec;
begin
 Client_IO.Open (File => Backup_File,
 Mode => Client_IO.In_File,
 Name => File_Name);
 -- Clear old client tree
 Phone_Tree.Clear (Client_Tree);
 loop -- Each iteration, one client record is restored to the tree
 exit when Client_IO.End_Of_File (Backup_File);
 -- Get the client record
 Client_IO.Read (File => Backup_File, Item => Client);
 -- Add it to the tree
 Phone_Tree.Insert (Tree => Client_Tree, Item => Client);
 end loop;

```

```

 Client_IO.Close (Backup_File);
end Restore;

end Clients;

```

Our Medical Response application contains three levels of abstractions. The program deals with the input of commands and data (its implementation is left as a programming exercise). The Clients ADO deals with a collection of client records. And the binary search tree organizes client records in a manner that optimizes the Find operation. The specifications of Clients and Binary\_Search\_Tree provide the interfaces between the different levels of abstractions.

The procedures Find, Interval, Add, Delete, and Change in Body 12-1 all contain exception handlers for exceptions that the binary search tree package might raise. These exception handlers translate the exceptions raised by the binary search tree package into exceptions meaningful to the program using the Clients ADO. For example, if Phone\_Tree.KEY\_ERROR is raised when we try to retrieve a client record from the binary search tree, procedure Find translates this exception to MISSING\_PHONE which is propagated to the application program. Why not just let Phone\_Tree.KEY\_ERROR propagate to the application program? Phone\_Tree.KEY\_ERROR is not visible outside of the body of our Clients ADO. Thus, the application program cannot provide an exception handler *specifically* for it.\*

### The Implementation Level

In this section we develop the algorithms for the binary search tree operations and implement them as a linked structure whose nodes are allocated dynamically. Here are the declarations for the private part of Specification 12-1. Each node contains an element and two pointers that designate the left and right subtrees of the node.

```

private

type Node_Type; -- Incomplete type declaration

type Tree_Type is access Node_Type; -- Access to a node
subtype Node_Ptr is Tree_Type; -- A synonym for our access type

type Node_Type is -- Complete type declaration
 record
 Info : Element_Type; -- One element
 Left : Node_Ptr; -- Link to left child
 Right : Node_Ptr; -- Link to right child
 end record;

end Binary_Search_Tree;

```

---

\*The application program can use an `others` exception handler to deal with *any* exception, even an exception whose name is unknown.

## Recursive Binary Search Tree Operations

Because the binary search tree is inherently a recursive structure, we first implement the algorithms using recursive solutions. In a later section we develop iterative Insert and Delete operations.

### The Size Operation

When we developed function `Factorial` in Chapter 11 we said that we could determine the factorial of  $N$  if we know the factorial of  $N - 1$ . The analogous statement here is that we can determine the number of nodes in the tree if we know the number of nodes in the left subtree and the number of nodes in the right subtree. This gives us the general case for our `Size` operation.

```
Result := 1 + Size (Tree.all.Left) + Size (Tree.all.Right);
```

Do you see where the 1 comes from? It's the root node.

What about the base case? We need a tree whose size is trivial to determine. The simplest tree is an empty tree. The size of an empty tree is zero. Here then is a recursive algorithm with our base and general cases:

### Size

```
if Tree is empty then
 Result := 0;
else
 Result := 1 + Size (Tree.all.Left) + Size (Tree.all.Right);
end if;
return Result;
```

Let's use our Three-Question Method to verify this recursive function.

1. *The Base-Case Question:* The base case occurs when the tree is empty. `Size` is then assigned a value of zero—which is the correct value for a tree with no nodes!—and no further (recursive) calls to `Size` are made, so the answer is yes.
2. *The Smaller-Caller Question:* To answer this question we must look at the parameters passed in the recursive call. There are two recursive calls in function `Size`. The first recursive call passes `Tree.all.Left` and the second passes `Tree.all.Right`. Each recursive call is made with a subtree that is *at least* one node smaller than the current tree. Eventually, the subtree passed will be an empty tree. At this point, as we verify with the base-case question above, we have reached the smallest case, and no further recursive calls are made. So the answer is yes.
3. *The General-Case Question:* In the case of a function like `Size`, we need to verify that the formula we are using actually results in the correct solution. Assuming that the recursive call `Size (Tree.all.Left)` gives us the correct value of the size of the left subtree and the recursive call `Size (Tree.all.Right)` gives us the correct value of the size of the right subtree, we calculate the size of the tree as the sum of the sizes of its two subtrees plus one. This is correct for a nonempty tree. In answering the first question, we have already ascertained that the function works for an empty tree. So the answer is yes.

When developing recursive tree algorithms, novice programmers frequently choose to use another base case: a tree with one node. This base case seems simple enough, but it inevitably leads to more complicated algorithms than the empty tree base case. When using recursion, always search for the simplest base case. For trees, this is the empty tree.

### The Retrieve and Modify Operations

At the beginning of this chapter we demonstrated how to search for an element in a binary search tree. First check to see if the item searched for is in the root. If it's not, compare the desired key with that of the root and then search either in the left or right subtree. This statement looks recursive and suggests the following recursive algorithm:

#### Retrieve

```

if Desired Key = Key of Root node then
 Element := Tree.all.Info -- Return the information in the root node
elsif Desired Key < Key of Root node
 Retrieve from Left Subtree
else
 Retrieve from Right Subtree
end if;
```

Because the left or right subtree is at least one node smaller than the original tree, the size decrease with each call. With our base case and smaller caller questions answered, let's look at the general case question. Any search has two possible outcomes: we find the key or we don't. Our base case applies when we find the desired key. What if the key is not in the tree? Then we recurse until we attempt to search an empty tree. This suggests an additional base case. If the tree we are searching is empty, the key is not to be found. Here is the Ada code for our recursive algorithm with both base cases:

```

procedure Retrieve (Tree : in Tree_Type;
 Key : in Key_Type;
 Element : out Element_Type) is
-- Recursive version
begin
 if Tree = null then
 raise KEY_ERROR; -- Base case
 elsif Key = Key_Of (Tree.all.Info) then
 Element := Tree.all.Info; -- Base case
 elsif Key < Key_Of (Tree.all.Info) then
 -- General case (search Left subtree)
 Retrieve (Tree => Tree.all.Left, Key => Key, Element => Element);
 else
 -- General case (search Right subtree)
 Retrieve (Tree => Tree.all.Right, Key => Key, Element => Element);
 end if;
end Retrieve;
```

Let's trace this algorithm using the tree in Figure 12-6. We want to retrieve the element with the key 18. The tree is not empty and 18 is greater than the key of the root (17),

so we call the procedure recursively, retrieving from the right subtree. This subtree is not empty and 18 is less than the key of its root (20), so we call the procedure recursively, retrieving from the left subtree. This subtree is not empty, and 18 is equal to the key of its root. Recursion halts and we return the information associated with this node.

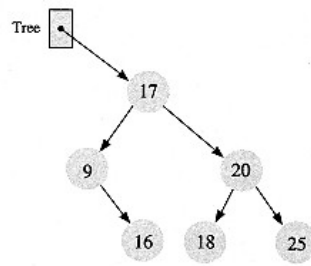


Figure 12-6  
Tracing the Retrieve Operation

Next, let's look at an example where the key is not found in the tree. We want to retrieve the element with the key 7 from the tree shown in Figure 12-6. The tree is not empty and 7 is less than the key of the root (17). We call this procedure recursively, retrieving from the left subtree. This subtree is not empty and again 7 is less than the key of its root (9), so we call the procedure recursively, retrieving from the left subtree. But this tree is empty so we raise the exception `KEY_ERROR`. This exception is propagated back to the original caller of procedure `Retrieve` who should handle it, perhaps by informing the user that the item was not found and asking the user to enter another value. In our Medical Response application, the Clients ADO translates the `KEY_ERROR` exception to `MISSING_PHONE`, a more meaningful name for the application program.

The code for `Modify` is nearly identical to `Retrieve`. Besides the different name and number of parameters, it requires a single change: `Tree.all.Info` is set to `Element` rather than the other way around.

### The Insert Operation

To create and maintain the information stored in a binary search tree, we must have an operation that adds new nodes into the tree. We use the following approach. A new node is always inserted into its appropriate position in the tree *as a leaf*. Figure 12-7 shows a series of insertions into a binary search tree.

Now let's develop an algorithm for the insert operation. From Figure 12-7 we see that the simplest insertion is the first one—inserting an item into an empty tree. As with nearly all recursive tree algorithms, *the empty tree is a base case*. And the general case nearly always involves repeating the algorithm on one of the two child nodes that are each the root of a subtree. Here is our algorithm:

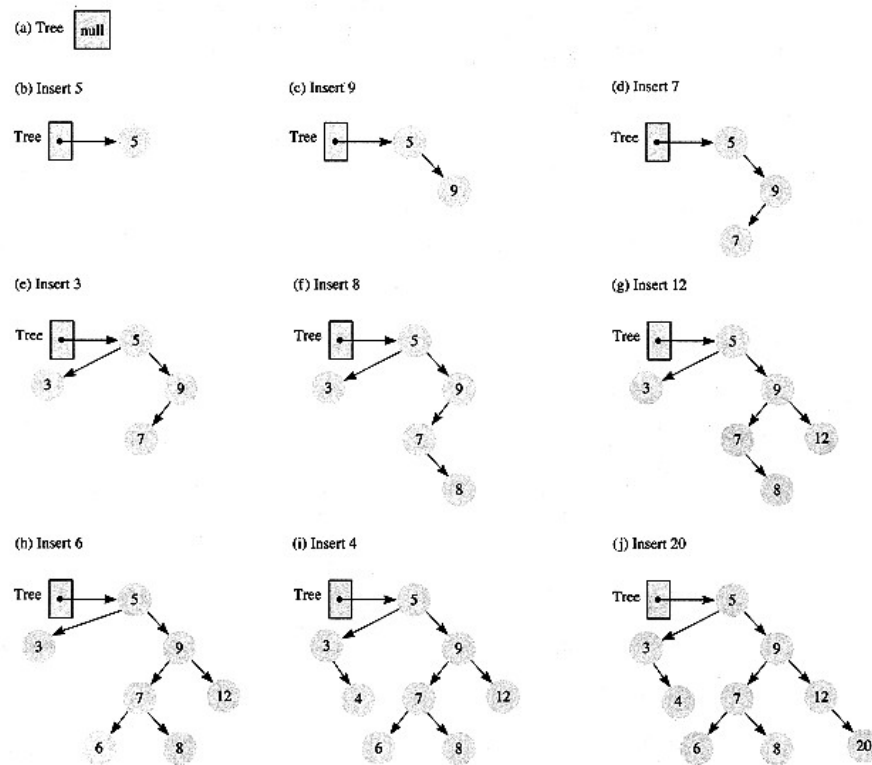


Figure 12-7  
Insertions into a Binary Search Tree

### Insert

```

if Tree is empty then
 Create a new (root) node containing the Item
else
 Insert into the appropriate subtree
end if

```

The appropriate subtree is determined by comparing the key of the Item to the key of the root node. If the Item's key is smaller, we insert it into the left subtree and if the Item's key is larger, we insert it into the right subtree.

Our specification for this operation indicates that the procedure should raise the exception `DUPLICATE_KEY` if a node with the same key as `Item` is in our tree. To accomplish this task, we can, as we did in procedure `Retrieve`, check to see if the root node contains the key of interest. If it does, we raise the exception. So like `Retrieve`, `Insert` has two base cases and two general cases. Here is the Ada code:

```
procedure Insert (Tree : in out Tree_Type;
 Item : in Element_Type) is
-- Recursive version
begin
 if Tree = null then
 -- Base case
 Tree := new Node_Type'(Info => Item, Left => null, Right => null);
 elsif Key_Of (Item) = Key_Of (Tree.all.Info) then
 -- Base case
 raise DUPLICATE_KEY;
 elsif Key_of (Item) < Key_Of (Tree.all.Info) then
 -- General case: Insert into left subtree
 Insert (Tree => Tree.all.Left, Item => Item);
 else
 -- General case: Insert into right subtree
 Insert (Tree => Tree.all.Right, Item => Item);
 end if;
end Insert;
```

Let's trace this procedure. In Figure 12-8(a), we want to insert an `Item` with the key value 13 into the tree whose root node is the node designated by `Tree`. Because 13 is greater than 7, we know that the new node belongs in the root node's right subtree. We now have defined a smaller version of our original problem: We want to insert a node with the key value 13 into the tree whose root is designated by `Tree.all.Right`. `Insert` is called recursively.

Procedure `Insert` begins its second execution, looking for the place to insert `Item` in the tree whose root is the node with the value 15 [Figure 12-8(b)]. We compare the key of `Item` (13) with the key of the root (15); 13 is less than 15, so we know that `Item` belongs in the tree's left subtree. Again we have redefined a smaller version of the problem. We want to insert a node with the key value 13 into the tree whose root is designated by `Tree.all.Left`.

Procedure `Insert` begins its third execution. We compare the key of `Item` (13) to the key of the root node (10) [Figure 12-8(c)], then call `Insert` to insert `Item` into the correct subtree. This time it's the subtree designated by `Tree.all.Right`.

At the beginning of the fourth execution of `Insert` we find that `Tree` is null. We have reached a base case. Figure 12-8(d) shows where we create the new node with the call:

```
Tree := new Node_Type' (Info => Item , Left => null, Right => null);
```

We are now finished with our insertion.

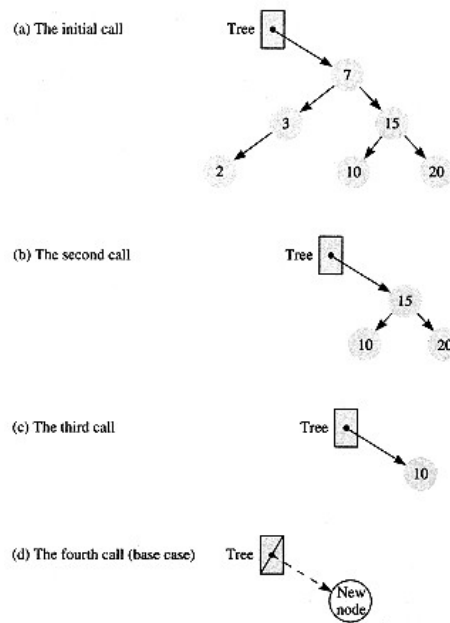


Figure 12-8  
The Recursive Insert Operation—Insert 13

Wait a minute! How does this assignment statement link the new node to the existing tree? To understand this point, you must understand that in every recursive execution the `Tree` parameter is actually a pointer field *within* the tree. Consider the formal and actual parameters of our recursive call. The last recursive call [Figure 12-9(a)] is

```
Insert (Tree => Tree.all.Right, Item => Item);
```

`Tree` is an *in out* parameter. Therefore the value of `Tree.all.Right` (null) is copied into `Tree` when the procedure is called [Figure 12-9(b)]. Next, the assignment statement changes the value of `Tree` to one that designates the newly allocated node [Figure 12-9(c)]. Now that the fourth execution of the procedure is complete we return to the caller. Because the formal parameter, `Tree`, is mode *in out*, its value is copied to the actual parameter [Figure 12-9(d)], linking the node with key 10 to the new node through its Right pointer.



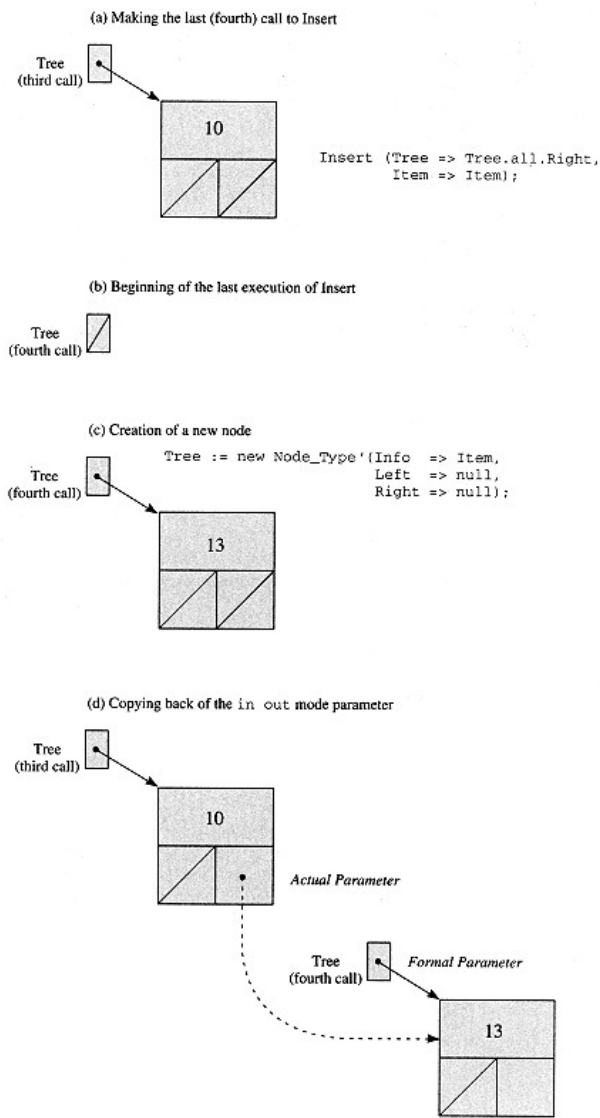


Figure 12-9  
The Formal Parameter Tree Copied to the Actual Parameter Tree.all.Right

### Insertion Order and Tree Shape

Earlier in this chapter we showed that the height of a tree is the critical factor in determining how efficiently we can search for elements. Let's look at another example. Figure 12-10 shows three different arrangements of the keys A through G. All of these trees are binary search trees. Searching the minimum-height tree shown in Figure 12-10(a) requires at most three calls to procedure `Retrieve`. But searching the maximum-height tree in Figure 12-10(c) may take as many as ten calls. As the number of nodes in a binary tree grows so does the difference between the heights of the minimum- and maximum-height trees (see the table on page 536).

|                                                                                                                                                         |
|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Skewed Tree</b>                                                                                                                                      |
| A subjective term describing the shape of a binary tree. The height of a skewed tree is closer to the maximum-height tree than the minimum-height tree. |
| <b>Bushy Tree</b>                                                                                                                                       |
| A subjective term describing the shape of a binary tree. The height of a bushy tree is closer to the minimum-height tree than the maximum-height tree.  |

Because nodes are always added as leaves, the order in which nodes are inserted determines the shape of the tree. Figure 12-10 illustrates how the same data, inserted in different orders, produces very differently shaped trees. If the values are inserted in order (or in reverse order), the tree is very **skewed** [Figure 12-10(c)]. A random mix of the elements produces a shorter, **bushy** tree [Figure 12-10(b)].

Because the height of the tree determines the maximum number of comparisons in a binary search, the tree's shape is very important. Obviously, minimizing the height of the tree maximizes the efficiency of the search. There are algorithms to adjust a tree to make its shape more desirable. The trees produced by these shape adjustments are called *balanced trees*. We do not discuss balancing algorithms in this text; they are subject matter for more advanced courses.

In the Medical Response Application discussed earlier in this chapter, procedure `Backup` (in Body 12-1) writes all the elements in a tree to disk. Later, procedure `Restore` may be used to build a new tree by reading these elements and inserting them into an initially empty tree. To prevent `Restore` from building a highly skewed tree like the one shown in Figure 12-10(c), `Backup` must *not* write the elements in order. Instead the elements in the tree are written to the disk in Preorder. When these elements are later read and inserted by `Restore` into a new tree, the original tree is reproduced. We use Figure 12-10(a) to illustrate this behavior. Traversing this tree in Preorder writes the elements in the order

D B A C F E G

When procedure `Restore` inserts the elements in this order into an empty tree it produces a tree identical to the original one. To convince yourself of this tree duplication, carry out the insertions by hand.

### The Delete Operation

The Delete operation is given the external pointer to a binary search tree (`Tree`) and the key of an element to delete (`Key`). This operation finds and deletes the node containing that key from the tree. The operation raises the exception `KEY_ERROR` if no node in the tree contains the given key. From our experience with procedure `Retrieve`, we know how to search recursively for a node with a given key. We can use a very similar algorithm for procedure `Delete`.

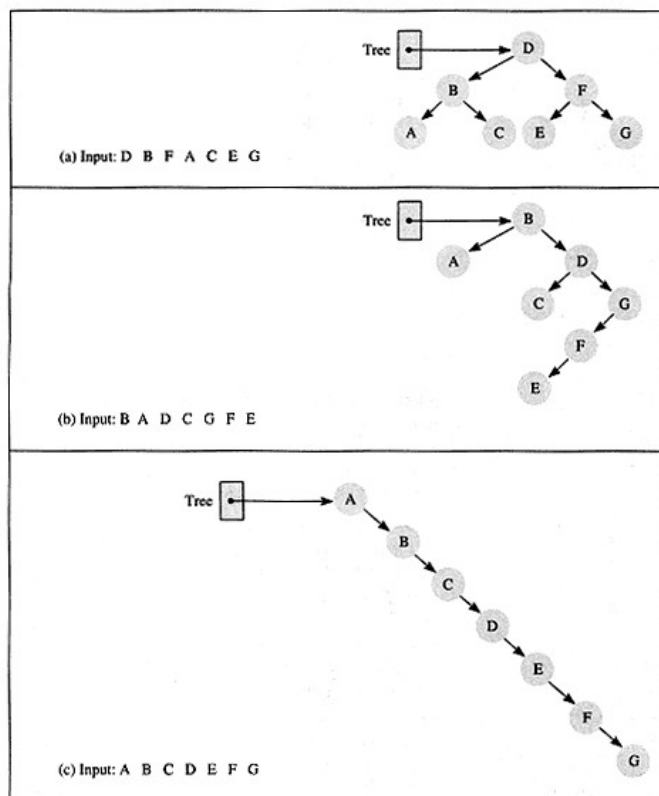


Figure 12-10  
The Input Order Determines the Shape of the Tree

```

procedure Delete (Tree : in out Tree_Type;
 Key : in Key_Type) is
-- Recursive version
begin
 if Tree = null then
 raise KEY_ERROR; -- Base case
 elsif Key = Key_Of (Tree.all.Info) then
 Delete_Root (Tree); -- Base case
 elsif Key < Key_Of (Tree.all.Info) then
 Delete (Tree => Tree.all.Left, Key => Key); -- General case
 else
 Delete (Tree => Tree.all.Right, Key => Key); -- General case
 end if;
end Delete;

```

There is only one difference between this procedure and the one we developed for `Retrieve`: We delete the root node instead of returning its `Info` field.

The delete task clearly involves changing pointers of the *parent* of the node we wish to delete. Because our recursive algorithms pass `Tree` as an `in out` mode parameter, the external pointer `Tree` is equivalent to the pointer in the parent that we must change (recall Figure 12-9).

We can break down the `Delete_Root` procedure into three cases, depending on the number of children the root node has.

1. *No children*: The root node is a leaf. As shown in Figure 12-11, deleting a root node with no children is simply a matter of setting `Tree` to `null` and then deallocating the memory used by the node.
2. *One child*: The simple solution of setting `Tree` to `null` doesn't suffice for removing a root node with children, because we don't want to lose all of its descendants from the tree. As illustrated in Figure 12-12, we want to make `Tree` skip over the deleted node and point instead to the child of the node we intend to delete. We then deallocate the memory used by the unwanted node.
3. *Two children*: This case is the most complicated because we can't make `Tree` designate *both* of the deleted node's children. There are several ways to accomplish this removal. The method we use is to replace the `Info` field of the root node with the `Info` field of its logical predecessor—the node whose key is closest in value to, but less than, the key of the root node. The logical predecessor is the node in the left sub-tree with the largest key. For example, in Figure 12-13 the logical predecessor of H is E, the logical predecessor of S is P, and the logical predecessor of L is J. This replacement node is guaranteed to have either no children or just a left child (see nodes E, P, and J in Figure 12-13). After replacing the `Info` in our root node with the `Info` in the logical predecessor, we delete the logical predecessor.

Figure 12-14 shows the deletion of a root node with two children. The logical predecessor of Q is P. After copying P into the root node, the original node containing P is unlinked from the tree by changing P's parent to designate P's only child. Finally the memory used by P is reclaimed.

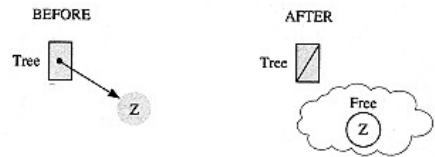


Figure 12-11  
Deleting a Root Node with No Children

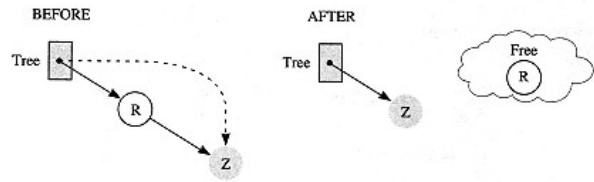


Figure 12-12  
Deleting a Root Node with One Child

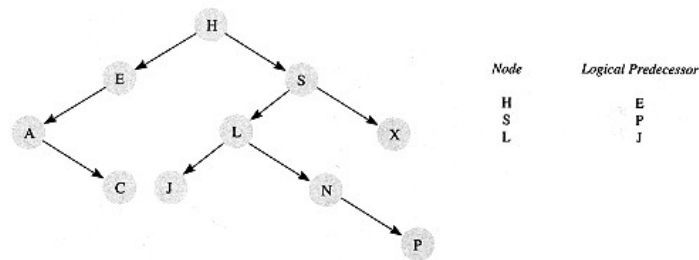


Figure 12-13  
The Logical Predecessors of Nodes with Two Children

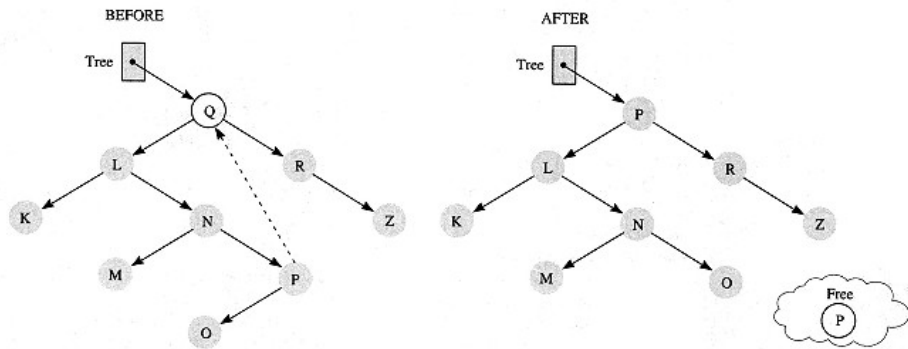


Figure 12-14  
Deleting a Root Node with Two Children

Here is the Ada code that implements these three cases. In this code, we have split the case of the root node with one child into two cases: having a right child and having a left child.

```

procedure Delete_Root (Tree : in out Tree_Type) is
-- This procedure deletes the root node from the given tree.
-- No recursion or loops used in this procedure.

-- Preconditions: Tree is not empty.

-- Postconditions: The root node of Tree is deleted.
-- Tree remains a binary search tree.

 To_Recycle : Node_Ptr; -- For recycling nodes
 Pred_Ptr : Node_Ptr; -- Designates the root's logical predecessor.
begin
 if Tree.all.Left = null and Tree.all.Right = null then
 -- Root node has no children.
 Free (Tree); -- Entire tree now empty
 elsif Tree.all.Left = null then
 -- Root node has only a right child.
 To_Recycle := Tree; -- Save for later deallocation
 Tree := Tree.all.Right; -- Unlink the root node
 Free (To_Recycle); -- Deallocate former root node
 elsif Tree.all.Right = null then
 -- Root node has only a left child.
 To_Recycle := Tree; -- Save for later deallocation
 Tree := Tree.all.Left; -- Unlike the root node
 Free (To_Recycle); -- Deallocate former root node
 else -- Root node has two children.
 -- Find and unlink the logical predecessor (The largest key in
 -- the left subtree).
 Find_And_Unlink_Max (Tree => Tree.all.Left,
 Max_Ptr => Pred_Ptr);
 Tree.all.Info := Pred_Ptr.all.Info; -- Copy Info from predecessor
 Free (Pred_Ptr); -- Deallocate predecessor
 end if;
end Delete_Root;

```

The first case, where the root node is a leaf, is straightforward. We simply free the memory used by the root node. Because *Tree* is an *in out* parameter, *null* is copied back to the actual parameter.

When the root has a single child, that child becomes the new root. *Tree* is changed to designate the non-null child. A different assignment statement is required depending on whether the child is a left child or a right child. In either case, the memory used by the original root node is deallocated.

When the root has two children we must find its logical predecessor. Recall that the logical predecessor of a node with two children *is the node with the maximum key in the left subtree* (see Figure 12-13 and 12-14). Procedure *Find\_And\_Unlink\_Max* is called to find the node in the root's left subtree with the maximum key value. This procedure returns a pointer to the maximum value node and unlinks it from the tree. *Delete\_Root* then copies the information and deallocates the memory.

At the expense of some clarity, we can improve the efficiency of `Delete_Root` by removing one of the tests. Notice that the action taken when there is only a right child also takes care of the case in which there are no children. When the left child is `null`, the right child gets stored in `Tree`. If the right child is also `null`, the `null` is stored in `Tree`, which is what we want if they are both `null`. We leave this modification as an exercise.

This only leaves procedure `Find_And_Unlink_Max`. This procedure finds the node in the given tree with the maximum key. The maximum value in any tree is in its *rightmost node*. To find it, we start at the root and keep moving down the right until the right child is `null`. For example, in Figure 12-15, we start at the root node, M, and move down the right links until we reach the node containing T, the largest key in the tree. Figure 12-15 also illustrates how the largest key node is unlinked from the tree. Its parent's pointer is changed so that it points the largest key node's left child. If the largest key node has no left child, `null` is assigned.

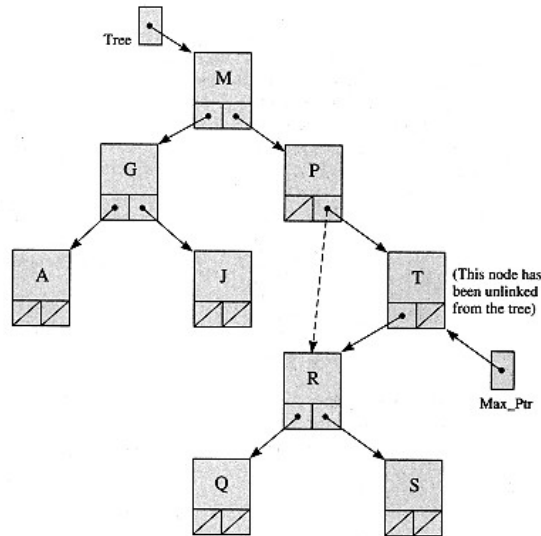


Figure 12-15  
Find\_And\_Unlink\_Max

Earlier we said that we are guaranteed that the logical predecessor of a root node with two children has no children or just a left child, and now you see why. The logical predecessor is the node with the *maximum* key in the left subtree. As we just showed, the node in a tree with the maximum key has no right child.

Here is the code for procedure `Find_And_Unlink_Max`:

```
procedure Find_And_Unlink_Max (Tree : in out Node_Ptr;
 Max_Ptr : out Node_Ptr) is
-- Finds and unlinks the node with the maximum key from the tree
-- Recursive version
```

```

-- Preconditions : Tree is not empty

-- Postconditions : Max_Ptr designates the node containing the
-- largest Key in Tree
-- The node designated by Max_Ptr is unlinked
-- from the tree
begin
 if Tree.all.Right = null then -- Is there a right child?
 -- Base case, root contains the maximum key in Tree
 Max_Ptr := Tree; -- Return pointer to it
 Tree := Tree.all.Left; -- Unlink it from Tree
 else
 -- General case, keep looking in the right subtree
 Find_And_Unlink_Max (Tree => Tree.all.Right, Max_Ptr => Max_Ptr);
 end if;
end Find_And_Unlink_Max;

```

The base case for this recursive procedure is that the root has no right child. When we find the maximum key, we unlink it from the tree by changing `Tree`. By changing the in out parameter `Tree`, we change the corresponding actual parameter—the parent's pointer. The general case in this procedure is to search to the maximum node in the right subtree.

Figure 12-16 shows five examples of deletions from a binary search tree. The first key deleted is J; it has no children so we simply set the right pointer of node H to null and free the memory used by A [Figure 12-16(b)]. Next we delete C; this node has a single child, A. We change node D's left pointer to designate this child and free the memory used by C [Figure 12-16(c)]. Next we delete L; this node has two children. First we find its logical predecessor, H. Then we unlink H from the tree by setting node D's right pointer designate H's left child, F. Finally we copy H into the root node and free the node that contains the original H [Figure 12-16(d)].

Using the remaining drawings in Figure 12-16, write down, as we did in the previous paragraph, the steps for the remaining two deletions. Then use all of these examples to trace the code we developed for procedure `Delete`.

### The Traverse Operation

We gave definitions for Inorder, Preorder, and Postorder tree traversals earlier in the chapter. These recursive definitions are easily implemented as recursive procedures. Here, for example, is the Ada code for an Inorder traversal:

```

procedure Inorder (Tree : in Tree_Type) is
begin
 if Tree /= null then
 Inorder (Tree => Tree.all.Left); -- Traverse left subtree Inorder
 Process (Tree.all.Info); -- Process the node
 Inorder (Tree => Tree.all.Right); -- Traverse left subtree Inorder
 end if;
end Inorder;

```

The base case of this recursive procedure, like nearly all recursive tree procedures, is an empty tree. In a traversal, we do nothing when the tree is empty.

Our binary search tree package included a generic procedure, `Traverse`, that has a parameter to select in which order the given tree is traversed. Procedure `Inorder` could be used in the body of `Traverse` to carry out one of the three possible orders. However,



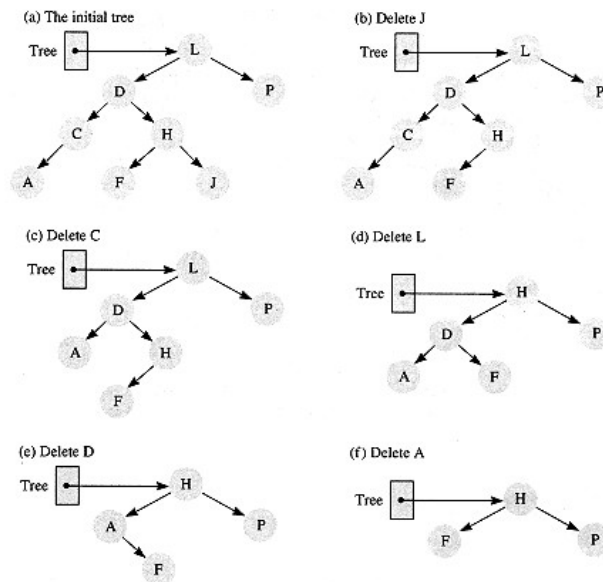


Figure 12-16  
Deletions from a Binary Search Tree

to meet the specifications, we must add the checks to ensure that the `Process` procedure supplied by the user does not change the key of the element. We leave this addition and the code for the other traverse orders as an exercise.

### The Clear Operation

The `Clear` operation is given the pointer to a binary search tree and destroys all the nodes, leaving the tree empty. To delete all the elements we traverse the tree, deleting each node from the tree when we process it. Which traversal order would be best? Although any of the traversal orders would result in the procedure performing correctly, one traversal order is more efficient than the others. Knowing that the `Delete` operation does less work to delete a leaf node than a node with children, we want to delete leaves first. If you delete the nodes in Postorder, each node is a leaf by the time of its turn to be deleted. And to delete a leaf, we just need to set the pointer in its parent to `null` and free the memory it uses. So the code for procedure `Clear` is a Postorder traversal with the process being a call to procedure `Free`. Here is the code:

```
procedure Clear (Tree : in out Tree_Type) is
begin
 if Tree /= null then
 Clear (Tree.all.Left); -- Delete all nodes in the left subtree
 Clear (Tree.all.Right); -- Delete all nodes in the right subtree
 Free (Tree); -- Delete the root node
 end if;
end Clear;
```

### Iterative Binary Search Tree Operations

The recursive `Retrieve`, `Modify`, `Insert`, and `Delete` procedures we have developed all use *tail recursion*; the recursive call is the last statement executed. As we discussed in Chapter 11, tail recursion can be converted to iteration without a programmer-defined stack. In this section we develop the iterative implementations of these four operations.

#### Searching a Binary Search Tree

In the recursive versions of the tree operations, we embedded the search function within the procedures that need it. The other alternative is to have a general search procedure as we did in Chapter 8 for the List. Let's do that here. Procedure `Search_Tree` is given a pointer to a binary search tree (`Tree`) and a key to find (`Key`). It returns a pointer to the desired node (`Location`) and a pointer to the node's parent (`Parent`) if an item with the given key is found. These two access variables are the tree equivalents of `Location` and `Pred_Loc` we used in Chapter 8 to search a list.

What do we do if we do not find an item with the given key, as in the case when we are inserting an element? For the List search, we used a Boolean parameter (`Found`) to indicate the results of the search. For this search we use the `Location` parameter to give us this information. If the search is not successful, `Location` is set to `null`; otherwise, it designates the node containing the key. Here is the declaration of our local search procedure:

```
procedure Search_Tree (Tree : in Tree_Type;
 Key : in Key_Type;
 Parent : out Node_Ptr;
 Location : out Node_Ptr);
-- Searches for the location of Key in the Tree

-- Preconditions : None

-- Postconditions : If Key is in Tree
-- Location designates the element with Key.
-- Parent designates the parent of the node
-- designated by Location. If Location designates
-- the root node of the tree, Parent is null.
-- else
-- Location is null.
-- If the tree is not empty then
-- Parent designates the node in the tree that
-- is logical parent of a node containing Key.
-- else
-- Parent is null.
```

Let's look at the search algorithm in detail. We use local equivalents (`Present` and `Previous`) of the out parameters `Location` and `Parent` to search the tree. Because we access the tree through its root, we initialize `Present` to the external pointer, `Tree`. We initialize `Previous` to `null`. We compare `Key` and `Key_Of (Present.all.Info)`. If the keys are equal, we have found the node we are looking for. If `Key` is less than `Key_Of (Present.all.Info)`, we look in the left subtree; if `Key` is greater than `Key_Of (Present.all.Info)`, we look in the right subtree. We use the "inchworm" method of moving the two external pointers down the tree.

## Search Tree

```

Present := Tree;
Previous := null;
loop
 exit when No More Nodes to Check or Key = Key_Of (Present.all.Info)
 if Key < Key_Of (Present.all.Info) then
 Previous := Present;
 Present := Present.all.Left; -- Go to left subtree
 else
 Previous := Present;
 Present := Present.all.Right; -- Go to right subtree
 end if;
end loop;

```

Like the iterative algorithms we developed to search a list, this algorithm has several loop termination conditions. Of course we stop searching when the desired key is found. We must also end this loop when there are no more nodes to check. We fall out of the tree when we move `Present` into an empty subtree. At this point, `Present` is null and `Parent` points to the node that would be the parent of the node with `Key`—if it did exist in the tree. (We use this value of `Parent` when we insert into a tree.) The resulting exit statement is

```

exit when Present = null or else Key = Key_Of (Present.all.Info);

```

This algorithm illustrates that the maximum number of comparisons in a binary search tree equals the height of the tree. As we discussed earlier, this may range from  $\log_2 N$  to  $N$  (where  $N$  is the number of tree nodes), depending on the shape of the tree.

Here is the procedure body that implements our search algorithm. We combined and moved the identical assignment statements in both branches of the if statement that update `Previous`. We have also taken advantage of Ada's initialization of all access variables to `null` to remove the assignment statement that explicitly initializes `Previous`.

```

procedure Search_Tree (Tree : in Tree_Type;
 Key : in Key_Type;
 Parent : out Node_Ptr;
 Location : out Node_Ptr) is
 Present : Node_Ptr; -- Local copies of
 Previous : Node_Ptr; -- out mode parameters
begin
 Present := Tree; -- Initialize pointer
 loop -- Each iteration, one node in the tree is checked
 exit when Present = null or else Key = Key_Of (Present.all.Info);
 Previous := Present; -- Move Inchworm tail
 if Key < Key_Of (Present.all.Info) then
 Present := Present.all.Left; -- Go to left subtree
 else
 Present := Present.all.Right; -- Go to right subtree
 end if;
 end loop;
 Parent := Previous; -- Copy results to
 Location := Present; -- out parameters
end Search_Tree;

```

Let's trace this algorithm using the tree shown in Figure 12-6. We want to find the element with key 18. `Present` is initially set as an alias of `Tree`, the external pointer. `Key` is greater than the key of the node designated by `Present`, so we advance the pointers. `Previous` now points to the root node and we move `Present` to the right; it now designates the node with the key 20. This key is greater than 18, and we move `Present` to the left: `Present` now designates the node with key 18 and `Previous` designates the node with key 20. Now `Key_Of (Present.all.Info) = 18` and we exit the loop. We exit the procedure with `Location` designating the node with the desired key and `Parent` designating this node's parent.

Next, let's look at an example where the key is not found in the tree. We want to find the node with key 7. `Present` is initially set as an alias of `Tree`. The key of the root node is greater than 7, so we move to the left. Now `Present` designates the node containing 9 and `Previous` designates the root node. Because 7 is less than 9, we move again to the left. Now `Present` equals `null`—it has fallen out of the tree. Because there's no more to search in this subtree, we exit the loop. We return from the procedure with `Location` equal to `null` and `Parent` designating the node containing 9. If we were calling `Search_Tree` with the intention of subsequently inserting a node with the key 7, we would now know two things:

1. Because `Location` is `null`, we know that there is no node with the key 7 in the tree.
2. Because `Parent` designates the last node visited before we fell out of the tree, we know that the new node, with a key value of 7, must be attached to the node designated by `Parent`. This information is very helpful when we are developing the iterative `Insert` procedure.

### The Insert Operation

The algorithm for the iterative `Insert` operation has the same three tasks that any insert operation must have:

#### Insert

```
Find the insertion place
Create a node to contain the new element
Link the new node into the structure
```

Creating a node is the same as in the recursive version of `Insert`—we use the allocator `new`. Finding the insertion point and linking the node into the structure are different.

Let's see how procedure `Search_Tree` can be used to perform the search for us. (To simplify our discussion, we use the `out` parameter names, `Location` and `Parent`, rather than their local equivalents, `Present` and `Previous`.) Suppose we want to insert an element with the key value 13 into the binary search tree pictured in Figure 12-17. In procedure `Search_Tree`, `Parent` is initialized to `null` and `Location` is initialized to designate the root of the tree [Figure 12-17(a)]. `Key` (13) is greater than the key of the root node (7) so we move `Location` to the right, dragging `Parent` along behind it [Figure 12-17(b)]. Now `Key` is less than the key of the node designated by `Location` (15), so we move `Location` to the left, with `Parent` following [Figure 12-17(c)]. `Key` (13) is now greater than the key of `Location.all.Info` (10), so `Parent` catches up, and `Location` moves to the right [Figure 12-17(d)]. At this point `Location` is `null`, so we exit the loop and return with the pointers as shown in Figure 12-17(d).

Of course, a node containing `Key` is not supposed to be found, for we are just now inserting its node. The good news is that `Location` has fallen out of the tree just at the spot

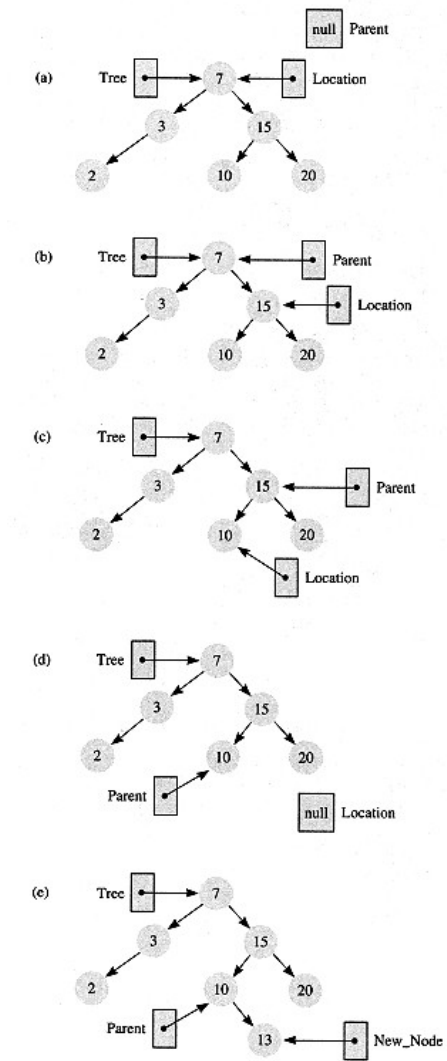


Figure 12-17  
Using Procedure Search\_Tree to Find the Insertion Point

where the new node should be inserted. Because `Parent` is trailing right behind `Location`, we can attach the new node to the node designated by `Parent` [Figure 12-17(e)].

Now we're ready to do the third task: fix the pointers in the node designated by `Parent` to attach the new node. In the general case we compare `Key` to the key of `Parent.all.Info`. Either the `Left` or the `Right` pointer field of this parent node must be set to designate the new node.

### Link in the New Node

```
if Key < Key_Of (Parent.all.Info) then
 Parent.all.Left := New_Node;
else
 Parent.all.Right := New_Node;
end if;
```

In the case of inserting the first node into an empty tree, however, `Parent` equals `null` and references using `Parent` raise `CONSTRAINT_ERROR`. We need to make inserting the first node into the tree a special case. We can test `Parent` to determine whether the tree is empty; if so, we change `Tree` to designate the new node. Taken together, the pieces of the insert design can be coded as the procedure body `Insert`.

```
procedure Insert (Tree : in out Tree_Type;
 Item : in Element_Type) is
-- Iterative version
 Parent : Node_Ptr; -- Designates new node's parent
 Location : Node_Ptr; -- Needed for search; should return as null
begin
 -- Search for insertion place
 Search_Tree (Tree => Tree,
 Key => Key_Of (Item),
 Parent => Parent,
 Location => Location);

 if Location /= null then -- Does Tree already contain Key?
 raise DUPLICATE_KEY;
 else -- Insert the new item
 if Parent = null then -- Special case, empty tree
 Tree := new Node_Type'(Info => Item,
 Left => null,
 Right => null);
 elsif Key_Of (Item) < Key_Of (Parent.all.Info) then
 -- Insert as left child
 Parent.all.Left := new Node_Type'(Info => Item,
 Left => null,
 Right => null);
 else
 -- Insert as right child
 Parent.all.Right := new Node_Type'(Info => Item,
 Left => null,
 Right => null);
 end if;
 end if;
end Insert;
```

## The Delete Operation

The same three cases exist for the iterative `Delete` procedure that exist for the recursive version: deleting a node with no children, one child, or two children. We can use `Search_Tree` to locate the node that we need to delete as well as its parent node. We performed the actual deletion in the recursive version with procedure `Delete_Root`. This procedure takes a single parameter—a pointer that designates the subtree whose root node is deleted. We can use this same procedure if we can determine the *place in the structure* to pass to `Delete_Root`. That is, given `Location` and `Parent`, we must determine whether the node we want to delete is the root node of the left subtree or the right subtree. If it is the left subtree, we pass `Parent.all.Left` to `Delete_Root`; otherwise, we pass `Parent.all.Right`.

```

procedure Delete (Tree : in out Tree_Type;
 Key : in Key_Type) is
-- Iterative version
 Location : Node_Ptr; -- Designates node to delete
 Parent : Node_Ptr; -- Designates parent of node to delete
begin
 -- Search for the node containing Key
 Search_Tree (Tree => Tree,
 Key => Key,
 Parent => Parent,
 Location => Location);

 if Location = null then -- Is the Key missing?
 raise KEY_ERROR;
 else -- Delete the node containing Key
 if Parent = null then
 -- Delete the root node
 Delete_Root (Tree => Tree);
 elsif Key < Key_Of (Parent.all.Info) then
 -- Delete the root node of the parent's left subtree
 Delete_Root (Tree => Parent.all.Left);
 else
 -- Delete the root node of the parent's right subtree
 Delete_Root (Tree => Parent.all.Right);
 end if;
 end if;
end Delete;

```

It is very important to recognize the difference between passing `Location` to `Delete_Root` and passing either `Parent.all.Left` or `Parent.all.Right`. See Figures 12-18 and 12-9. Nothing in the parent node is changed if we pass `Location` to `Delete_Root`.

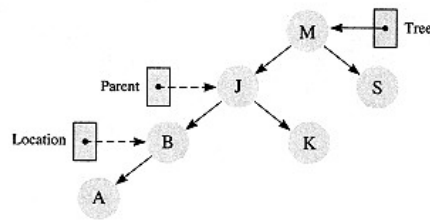


Figure 12-18  
Pointers Location and Parent Are External to the Tree

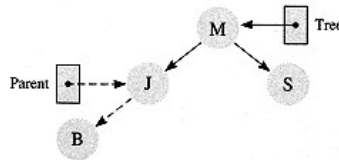


Figure 12-19  
Pointer Parent Is External to the Tree, but  
Parent.all.Left Is an Internal Pointer

We can use the procedure `Delete_Root` we have developed already (page 566) which contains no recursive calls. However, we used recursion in the procedure it calls to find a logical predecessor, `Find_And_Unlink_Max`. Because our recursive version uses tail recursion, it is not difficult to convert it to iteration. Here is an iterative version:

```

procedure Find_And_Unlink_Max (Tree : in out Node_Ptr;
 Max_Ptr : out Node_Ptr) is
 -- Finds and unlinks the node with the maximum key from the tree
 -- Iterative version.

 -- Preconditions : Tree is not empty.

 -- Postconditions : Max_Ptr designates the node containing the
 -- largest Key in Tree.
 -- The node designated by Max_Ptr is unlinked
 -- from the tree.

 Current : Node_Ptr; -- Local copy of out parameter
 Parent : Node_Ptr; -- Parent of Current node
begin
 Current := Tree; -- Start at the root
 loop
 exit when Current.all.Right = null;
 Parent := Current; -- Move inchworm tail
 Current := Current.all.Right; -- Move inchworm head
 end loop;
 Max_Ptr := Current; -- Copy result to out mode parameter

```



```

-- Unlink the largest key node from the tree
if Parent = null then
 Tree := Current.all.Left; -- Change external tree pointer
else
 Parent.all.Right := Current.all.Left; -- Change parent node
end if;
end Find_And_Unlink_Max;

```

The logic to unlink the node from the tree is a little more complex than our recursive version. We use an if statement to see if the node with the largest key has a parent in the tree. If it doesn't, the node with the largest key is the root of the subtree, and we have to change the external tree pointer to unlink it.

### The Retrieve and Modify Operations

Using our iterative `Search_Tree` procedure, these two operations are very easy to write. Here is the code for `Retrieve`:

```

procedure Retrieve (Tree : in Tree_Type;
 Key : in Key_Type;
 Element : out Element_Type) is
-- Iterative version
 Location : Node_Ptr; -- Designates node to retrieve
 Parent : Node_Ptr; -- Designates parent of node to retrieve
begin
 -- Search for the node containing Key
 Search_Tree (Tree => Tree,
 Key => Key,
 Parent => Parent,
 Location => Location);
 if Location = null then -- Is the Key missing?
 raise KEY_ERROR;
 else -- Return the information in the node
 Element := Location.all.Info;
 end if;
end Retrieve;

```

Again, the code for `Modify` is nearly identical to `Retrieve`. Besides the different name and number of parameters, it requires a single change: `Tree.all.Info` is set to `Element` rather than the other way around.

### Recursion or Iteration

Now that we have looked at both the recursive and iterative versions of `Insert` and `Delete`, can we determine which is better? In Chapter 11 we gave some guidelines for determining when recursion is appropriate. Let's apply these to the use of recursion with binary search trees.

*Is the depth of recursion relatively shallow?*

Yes. The depth of recursion is dependent on the height of the tree. If the tree is bushy, the depth of recursion is closer to  $O(\log_2 N)$  than to  $O(N)$ .

*Is the recursive solution shorter or clearer than the nonrecursive version?*

Yes. The recursive solutions are certainly shorter than the combination of the nonrecursive procedures plus the supporting procedure `Search_Tree`. Are they clearer?

Once you accept the fact that in every recursive execution the `Tree` parameter is actually a pointer field *within* the tree, the recursive versions become very easy to understand. The iterative versions are filled with special cases and if statements. And look at the complexity of the postconditions given for the local iterative procedure `Search_Tree`!

*Is the recursive solution much less efficient than the nonrecursive version?*

No. Both the recursive and nonrecursive versions of `Insert` and `Delete` are  $O(\log_2 N)$  operations assuming a bushy tree. But, as our timing experiments that we detail later in Table 12-2 shall show, the execution times of the recursive versions of `insert` and `delete` are noticeably but not significantly longer than their iterative counterparts on the computer on which these experiments were run.

We give the recursive versions of the `Insert` and `Delete` procedures an "A"; they score highly as good uses of recursion. The clarity they provide the programmer is worth the extra execution time required.

### Comparing Binary Search Trees to Linear Lists

A binary search tree is an appropriate structure for many of the same applications we have discussed previously in conjunction with other ordered list structures. The special advantage of using such a tree is that it facilitates *searching* while conferring the benefits of linking the elements. It provides the best features of both the sequential and linked lists: Like a sequential list, it can be searched quickly, using a binary search. Like a linked list, it allows insertions and deletions without having to move data physically. Thus it's particularly suitable for applications in which search time must be minimized or in which the nodes are not necessarily going to be processed in sequential order.

As usual, there is a trade-off. The binary search tree, with its extra pointer in each node, takes up more memory space than a singly linked list. In addition, the algorithms for manipulating the tree are somewhat more complicated. If all of the list's uses involve sequential rather than random processing of the elements, the tree may not be as good a choice.

Suppose we have 1000 customer records in a list. If the main activity in the application is to send out updated monthly statements to the customers, and if the order in which the statements are printed is the same as the order in which the records appear on the list, a linked list would be suitable. But suppose we decide to keep a terminal available to give out account information to the customers whenever they ask. If the data are kept in a linked list, the first customer on the list can be given information almost instantly, but the last customer has to wait while the other 999 records are examined. We need to examine about 10 records to find any customer in a binary search tree. When direct access to the records is a requirement, a binary search tree is a more appropriate structure.

### Big-O and Execution Time Comparisons

To illustrate the difference between the operations that manipulate a linked list and a binary search tree, we have run a timed test. Before we tell you how the tests came out, let's describe the operations in terms of their Big-O approximations. Then we can see how well we would expect the operations to perform.

Finding the node to process, as we would expect in a structure dedicated to searching, is the most interesting operation to analyze. In the best case—if the order in which the elements were inserted results in minimum-height tree—we can find any node in the tree with at most  $\lceil \log_2(N + 1) \rceil$  comparisons. We would expect to be able to locate a random element in such a tree much faster than finding an element in an ordered linear list. In the worst case—if the elements were inserted in order from smallest to largest or vice versa—the tree isn't really a tree at all; it is a linear list, linked through either the Left or Right pointer field. (This is called a "degenerate" tree.) In this case, the tree operations should perform much the same as the operations on a linked list.

The Insert, Modify, and Retrieve operations are basically the order of finding the node  $[O(\log_2 N)]$  plus tasks that are  $O(1)$ —for instance, creating a node, resetting pointers, or copying data. Thus these operations are all described as  $O(\log_2 N)$ . The Delete operation consists of finding the node plus `Delete_Root`. In the worst case (deleting a node with two children), `Delete_Root` must find the replacement value, an  $O(\log_2 N)$  operation. (Actually, the two tasks together add up to  $\log_2 N$  comparisons because if the delete node is higher in the tree, fewer comparisons are needed to find it, and more comparisons may be needed to find its replacement node.) Otherwise, if the deleted node has 0 or 1 child, `Delete_Root` is an  $O(1)$  operation. So Delete may also be described as  $O(\log_2 N)$ . Notice that this analysis is true for both the recursive and iterative versions. Finding the node to process is  $O(\log_2 N)$  whether the search is done using iteration or recursion.

Both the Traverse and Clear operations require the tree to be traversed, processing each element once. Thus these are  $O(N)$  operations. The orders of magnitude for the tree and list operations are compared in Table 12-1.

**Table 12-1** Big-O Comparisons of Tree and List Operations

|          | <i>Binary Search Tree</i> | <i>Linked List</i> |
|----------|---------------------------|--------------------|
| Clear    | $O(N)$                    | $O(N)$             |
| Traverse | $O(N)$                    | $O(N)$             |
| Retrieve | $O(\log_2 N)^*$           | $O(N)$             |
| Modify   | $O(\log_2 N)^*$           | $O(N)$             |
| Insert   |                           |                    |
| randomly | $O(\log_2 N)$             | $O(N)$             |
| in order | $O(N)$                    | $O(N)$             |
| Delete   | $O(\log_2 N)^*$           | $O(N)$             |

Now let's see how our test came out. For each structure (binary search tree and linear list), we inserted 1000 elements with integer keys and then deleted them one by one. We ran each test three times. In the first test, we inserted the elements in random order, to produce a bushy tree. We then deleted the elements one by one, in the same order in which they were inserted. As you can see in the Table 12-2, the results were as expected: both the tree insertions and deletions were faster than the comparable linear list operations.

In the second test, we inserted 1000 elements with the keys in order, and then delete them one by one in the same order. In this case, the tree produced essentially a linear list, and the tree operations were not as fast as the list operations. The overhead of recursion in this set of experiments is obvious. The depth of recursion in these highly skewed trees is great.

---

\*Assuming bushy trees.

In the third test, we inserted elements with the keys in reverse order, and then deleted them one by one in the same order. In this case, the list insertions were essentially  $O(1)$  operations, as we were always inserting at the front of the list. Again, the tree was a linear structure, this time leaning in the opposite direction. Because tree insertions are always made to leaf nodes, however, the insertions were much slower—we were always inserting the "last" node— $O(N)$  operations. The deletions worked in reverse: the first (root with one child) element was always deleted from the tree [an  $O(1)$  operation], but the last node was always deleted from the list [an  $O(N)$  operation]. The timing results are as we would expect.

Remember, the results of these experiments are only valid for the system on which they were run. Only Big-O allows us to compare algorithms without regard to what computer system we run them.

**Table 12-2** Results of the Timed Implementation Comparisons (in seconds)

|                          | Binary Search Tree |           | Linked List |
|--------------------------|--------------------|-----------|-------------|
|                          | Recursive          | Iterative |             |
| Case 1: Random           |                    |           |             |
| Insert                   | 0.190              | 0.138     | 0.717       |
| Delete                   | 0.098              | 0.086     | 0.674       |
| Case 2: In Order         |                    |           |             |
| Insert                   | 4.04               | 1.39      | 1.27        |
| Delete                   | 0.02               | 0.01      | 0.02        |
| Case 3: In Reverse Order |                    |           |             |
| Insert                   | 3.96               | 1.38      | 0.03        |
| Delete                   | 0.02               | 0.02      | 1.28        |

This exercise does illustrate an important point about binary search trees: The order in which the elements are inserted is critical in determining the efficiency of the tree operations. This order determines the shape of the tree—short and bushy (good) or tall and stringy (bad). The shorter the tree (fewer levels to search), the faster we can access *random* elements. The speed of the tree Traversal is unaffected by the shape. A short bushy tree, a tall stringy tree, or a linked list all take about the same time to traverse. However, there are space implications; the overhead from the recursive calls to traverse a tall, stringy tree may raise `STORAGE_ERROR` if the run-time stack overflows due to the large number of recursive calls required to reach the leaf nodes.

## Summary

In this chapter we have seen how the binary tree may be used to structure ordered information to reduce the search time for any particular element. For applications where direct access to the elements in an ordered structure is needed, the binary search tree is a very useful class. If the tree is bushy, we can access any node in the tree with an  $O(\log_2 N)$  operation. The binary search tree combines the advantages of quick random-access (like a binary search on a sequential list) with the flexibility of a linked structure.

We also saw that the tree operations could be implemented very elegantly and concisely using recursion. This makes sense, as a binary tree is itself best defined as a "recursive" structure: any node in the tree is the root of another binary tree. Each time we moved

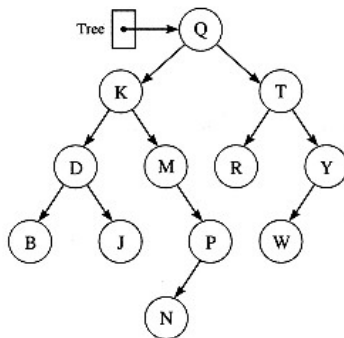
down a level in the tree, taking either the right or left path from a node, we cut the size of the (current) tree in half, a clear case of the smaller-caller. We also saw cases of iteration that replaced recursion (Insert and Delete).

In this chapter we stored all our tree values in dynamically allocated memory. We can, however, use the techniques we described in Chapter 10 to store our tree values in static structures (arrays and direct files). In the next chapter we examine another way of storing trees in static structures.

### Exercises

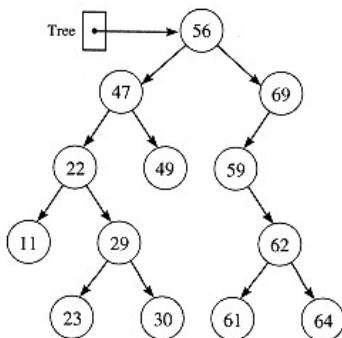
- What does the level of a binary search tree mean in relation to the searching efficiency?
  - What is the maximum number of levels that a binary search tree with 100 nodes can have?
  - What is the minimum number of levels that a binary search tree with 100 nodes can have?
- What is the maximum total number of nodes in a tree that has  $N$  levels? (Remember that the root is Level 0.)
  - $N^2 - 1$
  - $2^N$
  - $2^{N+1} - 1$
  - $2^{N+1}$
- What is the maximum number of nodes in the  $N$ th level of a binary tree?
  - $N^2$
  - $2^N$
  - $2^{N+1}$
  - $2^N - 1$
- How many ancestors does a node in the  $N$ th level of a binary search tree have?
- How many different *binary trees* can be made from three nodes that contain the key values 1, 2, and 3?
  - How many different *binary search trees* can be made from three nodes that contain the key values 1, 2, and 3?
- Draw all the possible binary trees that have four leaves and all the nonleaf nodes have two children.
- In the Medical Response application discussed in this chapter, why did we *not* have to declare and write an `Equal` function in package `Clients` for `phone_Rec` as we did for `Name_Rec`?

Answer the questions in Exercises 8–10 independently, using the following tree.



8. (a) What are the ancestors of node P?  
 (b) What are the descendants of node K?  
 (c) What is the maximum possible number of nodes in the tree at the level of node W?  
 (d) What is the maximum possible number of nodes in the tree at the level of node N?  
 (e) Insert node O. How many nodes would be in the tree if it were completely full down to and including the level of node O?
9. Show what the tree would look like after each of the following changes. (Use the original tree to answer each part.)  
 (a) Add node C.  
 (b) Add node Z.  
 (c) Add node X.  
 (d) Delete node M.  
 (e) Delete node Q.  
 (f) Delete node R.
10. Show what order the nodes in the tree are processed by  
 (a) an inorder traversal of the tree  
 (b) a postorder traversal of the tree  
 (c) a preorder traversal of the tree
11. Draw the binary search tree whose elements are inserted in the following order:  
 50 72 96 94 107 26 12 11 9 2 10 25 51 16 17 95

Exercises 12–16 use the following tree.



12. (a) What is the height of the tree?  
 (b) What nodes are on level 3?  
 (c) Which levels have the maximum number of nodes that they could contain?  
 (d) What is the maximum height of a binary search tree containing these nodes? Draw such a tree.  
 (e) What is the minimum height of a binary search tree containing these nodes? Draw such a tree.
13. (a) Trace the path that would be followed in searching for a node containing 61.  
 (b) Trace the path that would be followed in searching for a node containing 28.

14. Show what order the nodes in the tree are processed by
  - (a) an inorder traversal of the tree
  - (b) a postorder traversal of the tree
  - (c) a preorder traversal of the tree
15. Show how the tree would look after the deletion of 29, 59, and 47.
16. Show how the (original) tree would look after the insertion of nodes containing 63, 77, 76, 48, 9, and 10 (in that order).
17. *True or False?*
  - (a) Invoking the delete procedure in this chapter might create a tree with more levels than the original tree had.
  - (b) A preorder traversal processes the nodes in a tree in the exact reverse order that a postorder traversal processes them.
  - (c) An inorder traversal will always process the elements of a tree in the same order, regardless of the order in which the elements were inserted.
  - (d) A preorder traversal will always process the elements of a tree in the same order, regardless of the order in which the elements were inserted.
18. If you wanted to traverse a tree, writing all the elements to a file, and later (the next time you ran the program) rebuild the trees by reading and inserting, would an inorder traversal be appropriate? Why or why not?
19. (a) One hundred integer elements are chosen at random and inserted into an ordered linked list and a binary search tree. Describe the efficiency of searching for an element in each structure, in terms of Big-O.  
 (b) One hundred integer elements are inserted in order, from smallest to largest, into an ordered linked list and a binary search tree. Describe the efficiency of searching for an element in each structure, in terms of Big-O.
20. The key of the nodes in a binary search tree is a short character string.
  - (a) Show how such a tree would look after the following words were inserted (in the order indicated):  
 monkey canary donkey deer zebra yak walrus vulture penguin quail
  - (b) Show how the tree would look if the same words were inserted in this order:  
 quail walrus donkey deer monkey vulture yak penguin zebra canary
  - (c) Show how the tree would look if the same words were inserted in this order:  
 zebra yak walrus vulture quail penguin monkey donkey deer canary
21. Write the body for the generic traverse procedure specified in the binary search tree declaration given in this chapter.
22. (a) Make the modifications to the `Delete_Root` procedure suggested on page 567 to improve its efficiency.  
 (b) Which version of `Delete_Root` is easier to understand?
23. (a) Write a procedure called `Find_And_Unlink_Min` that finds the node with the smallest key value in a tree and unlinks it from the tree. A pointer to the unlinked node should be returned as an `out` parameter.

(b) Modify the `Delete_Root` procedure 566 from the chapter so that it uses the immediate successor (rather than the predecessor) of the value to be deleted in the case of deleting a node with two children. You should call the procedure `Find_And_Unlink_Min` that you wrote in part (a).

24. (a) Use the Three-Question method to verify the recursive version of `Insert`.

(b) Use the Three-Question method to verify the recursive version of `Delete`.

25. Procedure `Change_Key` modifies the key value of an element in a binary search tree. It changes the key of the node in the tree whose key is `Old_Key` to `New_Key`.

(a) Write the procedure declaration of `Change_Key`. Be sure to include the necessary comments to adequately specify the procedure. Be sure to include necessary exception comments.

(b) Write `Change_Key` as an application operation that does not know how the tree is implemented.

26. Write a nonrecursive generic procedure `Ancestors` that processes (using a user-supplied procedure) the ancestors of the node whose key is supplied. The ancestors should be processed from "oldest" to "youngest"—that is, process the parent before the child. You may assume that the given key is in the tree. Use the following procedure declaration:

```
generic
 Procedure Process (Element : in Element_Type);
 procedure Ancestors (Tree : in Tree_Type;
 Key : in Key_Type);
```

27. (a) Write a recursive version of procedure `Ancestors` described in Exercise 26.

(b) Write the Ada code necessary to instantiate from `Ancestors` a procedure to display the key of all the ancestors of a given node. You may assume that the key of the tree is type `Integer`.

(c) Write a call of the procedure you instantiated in part (b) to display the keys the ancestors of the node containing the value 27.

28. Write a recursive version of procedure `Ancestors` (see Exercise 26) that prints out the ancestors in reverse order (first the parent, then the grandparent, and so on).

29. The Binary Search Tree package is extended to include Boolean function `Is_BST`, which determines if a binary tree is a binary search tree.

(a) Write the declaration of function `Is_BST`. Include adequate comments.

(b) Write a recursive implementation of this function.

30. The Binary Search Tree package is extended to include functions that input a pointer to a binary search tree, and return the number of nodes that meet certain requirements.

(a) Write a recursive function `Leaf_Count`, which returns the number of leaf nodes in the tree.

(b) Write a recursive function `Single_Parent_Count`, which returns the number of nodes in the tree that have only one child.

31. Write the procedure `Big_Value_Count`, which returns the number of nodes in the tree whose key value is greater than `Value` (an input parameter to the procedure). Write `Big_Value_Count` as an application operation that does not know how the tree is implemented. Hint: Instantiate a procedure from `Traverse`.

32. The Binary Search Tree package is extended to include procedure `Copy`, which inputs a pointer to the root node of a binary search tree, and outputs a pointer to the root node of a duplicate tree.

(a) Write the declaration of procedure `Copy_Tree`. Include adequate comments.

(b) Write the body of procedure `Copy_Tree`.



33. The Binary Search Tree package is extended to include a Boolean function `Similar_Trees`, which inputs pointers to two binary trees and determines if the shapes of the trees are the same. (The nodes do not have to contain the same values, but each node must have the same number of children.)

- (a) Write the declaration of function `Similar_Trees`. Include adequate comments.
- (b) Write the body of function `Similar_Trees`.

34. Procedure `Mirror` inputs a pointer to the root node of a binary search tree and creates a mirror image of the tree.

- (a) Write the declaration of procedure `Mirror`. Include adequate comments.
- (b) Write the body of procedure `Mirror`.
- (c) Can the binary tree returned from this procedure be used for binary searching? If so, how?

Use the following declarations for Exercises 35 and 36.

```
type Integer_Array is array (Positive range <>) of Integer;

type List_Type (Max_Size : Positive) is
 record
 Elements : Integer_Array (1..Max_Size);
 Num_Elements : Natural := 0;
 end record;

function Identity (Element : in Integer) return Integer;

package Int_Tree is new Binary_Search_Tree (Element_Type => Integer,
 Key_Type => Integer,
 Key_Of => Identity,
 "<" => "<",
 "=" => "=");
```

35. (a) Write a procedure, `Make_Tree`, that creates a binary search tree from the elements in a sorted list of integers. You cannot traverse the list inserting the elements in order because that would produce a tree that would have  $N$  levels. You must create a tree with at most  $\lceil \log_2(N+1) \rceil$  levels.

- (b) Write a procedure called `Make_Bushy` that turns a binary search tree of integers into a minimum-height tree using the following algorithm.

#### **Make Bushy**

Create a sorted array of integers by traversing the tree  
 Clear the tree  
 Create a the minimum-height tree by calling procedure `Make_Tree`.

36. Write a Boolean function, `Matching_Elements`, that determines if a binary search tree and a sequential list contain the same values.

Use the following declarations in Exercises 37 and 38.

```
type Appraisal_Type is (Superior, Good, Fair, Warning, Pink_Slip);

type Employee_rec is
 record
 ID : Positive;
 Salary : Float;
 Appraisal : Appraisal_Type
 end record;
```

[illegible]

37. Instantiate a traversal to write the procedure `Raise_Time`, which is given a pointer to the root node of a binary search tree of employee records and modifies the salary of every employee according to his or her last appraisal rating:

|           |                             |
|-----------|-----------------------------|
| Superior  | raise current salary by 10% |
| Good      | raise current salary by 5%  |
| Fair      | raise current salary by 3%  |
| Warning   | no change                   |
| Pink Slip | set salary to 0             |

38. Instantiate a traversal to write `Display_By_Appraisal` that is given a pointer to the root node of a binary search tree and displays the IDs of every record with each Appraisal rating. First all the IDs of the Superior employees should be displayed (in order), then the IDs of the Good employees, and so on. Each set of IDs should be preceded by a label describing the Appraisal rating.

Use the following declarations in Exercises 39–41.

```
subtype Name_String is String (1..20);
type Sex_Type is (Male, Female);
type Class_Type is (Freshman, Sophomore, Junior, Senior, Grad);
type GPA_Type is digits 4 range 0.0..4.0;
```

```
type Student_Rec is
 record
 Last_Name : Name_String;
 First_Name : Name_String;
 ID_Num : Positive;
 Sex : Sex_Type;
 Class : Class_Type;
 GPA : GPA_Type;
 end record;
```

```
function GPA_Of (Student : in Student_Rec) return GPA_Type;
function ID_Of (Student : in Student_Rec) return Positive;
```

[illegible][illegible]

39. The dean wants to specially honor all of the students with a GPA of 4.0. Write the application procedure `Print_Genius` that takes a pointer to the binary search tree and prints the name of each student with a GPA of 4.0. The names can be printed in any order.

40. A women's organization wants to select some graduating female students to receive awards. Write the application level procedure `Print_Senior_Women` to print the name, GPA, and `ID_Num` of each of the women in the senior class, ordered from highest to lowest GPA. You may use the stack package developed in Chapter 6.

41. Write a procedure that takes an `ID_Num` and either displays all of the corresponding student information (if the specified student is found) or prints "No student found with" and then the `ID_Num`.

42. We have added the following function to Specification 12-1. Write its body. Hint: Use recursion.

```
function Height (Tree : in Tree_Type) return Natural;
-- Returns the number of levels in a binary search tree
--
-- Preconditions : None
--
-- Postconditions : Height = number of levels in Tree
```

Use these declarations for Exercises 43 and 44.

```
type Node_Ptr is 0..Max_Nodes;
```

```
Nil : constant Node_Ptr := 0;
```

```
type Node_Type is
 record
 Info : Character;
 Left : Node_Ptr := Nil;
 Right : Node_Ptr := Nil;
 end record;
```

```
subtype Index_Type is Node_Ptr range 1..Node_Ptr'Last;
type Node_Array is array (Index_Type) of Node_Type;
```

```
Nodes : Node_Array;
```

43. In Chapter 10 we discussed how a linked list can be stored in an array of nodes using index values as "pointers" and managing our list of free nodes. We can use these same techniques to store the nodes of a binary search tree in an array, rather than using dynamic storage allocation. Given the declarations above, show how the array would look after these elements had been inserted:

Q L W F M R N S

Free space is linked through the Left field. Be sure to fill in all the spaces. If you do not know the contents of a space, use "?".

|       |       |       |        |      |                          |
|-------|-------|-------|--------|------|--------------------------|
| Nodes | .Info | .Left | .Right | Tree | <input type="checkbox"/> |
| (1)   | Q     |       |        | Free | <input type="checkbox"/> |
| (2)   | L     |       |        |      |                          |
| (3)   | W     |       |        |      |                          |
| (4)   | F     |       |        |      |                          |
| (5)   | M     |       |        |      |                          |
| (6)   | R     |       |        |      |                          |
| (7)   | N     |       |        |      |                          |
| (8)   | S     |       |        |      |                          |
| (9)   |       |       |        |      |                          |
| ⋮     |       |       |        |      |                          |
| ⋮     |       |       |        |      |                          |

44. Show the contents of the array in Exercise 43 after 'B' has been inserted and 'R' deleted.

|       |       |       |        |      |                          |
|-------|-------|-------|--------|------|--------------------------|
| Nodes | .Info | .Left | .Right | Tree | <input type="checkbox"/> |
| (1)   |       |       |        | Free | <input type="checkbox"/> |
| (2)   |       |       |        |      |                          |
| (3)   |       |       |        |      |                          |
| (4)   |       |       |        |      |                          |
| (5)   |       |       |        |      |                          |
| (6)   |       |       |        |      |                          |
| (7)   |       |       |        |      |                          |
| (8)   |       |       |        |      |                          |
| (9)   |       |       |        |      |                          |
| ⋮     |       |       |        |      |                          |
| ⋮     |       |       |        |      |                          |

**Programming Problems**

- 1. Implement and test the body for the binary search tree ADT developed in this chapter
  - (a) using recursive implementations
  - (b) using iterative implementations for Insert and Delete

Turn in a copy of your test plan, a listing of your test program, a listing of your binary search tree body, and the output of your test program.

- 2. You are given the following procedure declaration:

```

generic
 type Element_Type is private;
 type Array_Type is array (Positive range <>) of Element_Type;
 with function "<" (Left : in Element_Type; Right : in Element_Type)
 return Boolean;

procedure Tree_Sort (Info : in out Array_Type);
-- Sorts the elements in List from smallest key to largest key
-- as determined by function "<"

-- Preconditions: None

-- Postconditions: The elements in Info are in order by function "<"

```

(a) Write the body of this procedure using the following algorithm:

### Tree Sort

```

 Insert all of the array components into a binary "search" tree
 Traverse the binary search tree in Inorder

```

The traverse process consists of copying the element into the array. The array `Info` may contain duplicate elements. Therefore you must modify the tree insert algorithm developed in this chapter.

- (b) Assuming that the elements in the array are originally in random order, what is the Big-O of this sort procedure?
  - (c) Assuming that the elements in the array are originally in order, what is the Big-O of this sort procedure?
  - (d) Assuming that the elements in the array are originally in reverse order, what is the Big-O of this sort procedure?
3. (a) Write a package declaration for a binary search tree ADO that stores the tree nodes in a direct file. Be sure to include an operation that saves the tree information necessary to use the tree again the next time the program is run.
- (b) Implement the package body for this ADO.
  - (c) Write a "dump" program that displays all of the components of the file produced by your ADO.
  - (d) Write a test plan for testing this ADO. You may use the dump program in your plan.
4. Using the `Clients` ADO and binary search tree ADT developed in this chapter, write and test a program to implement the Medical Response specification given on page 544.
5. In an inorder traversal, we move as far to the left as we can letting the run-time stack keep track of the nodes that we pass. When we reach a null tree, we back up, process the node, and move to the node's right child. The process then repeats itself. This algorithm can be implemented using a stack rather than using recursion. Write an iterative inorder traversal procedure. You may instantiate a stack from any of the packages developed in Chapter 6. (Hint: The traversal is finished when the stack is empty and the right child is `null`.)
6. Your assignment is to extend the Binary Search Tree class by adding the following operations:

| <i>Name</i>   | <i>Type</i>       | <i>Purpose</i>                                                                                                                                         |
|---------------|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| Similar_Trees | Function          | Determine if two binary trees have the same shape. (The nodes do not have to contain the same values, but they must have the same number of children.) |
| Leaf_Count    | Function          | Determine the number of leaf nodes in a binary search tree.                                                                                            |
| Height        | Function          | Determine the number of levels in a binary search tree.                                                                                                |
| Copy          | Procedure         | Create a copy of a binary search tree.                                                                                                                 |
| Ancestor      | Generic Procedure | Processes the ancestors of the tree node with the specified key.                                                                                       |
| Make_Bushy    | Procedure         | Reorganizes the nodes in the binary search tree so that it is a minimum height tree. (See Exercise 35.)                                                |

The first part of your assignment is to write the specifications of each operation as we normally do in a package specification. Be sure to include a brief description, preconditions, postconditions, and exceptions. Once you have specified the operations, you should implement them. Use recursion wherever possible.

### ***Testing the Operations***

Write a test plan describing the test cases needed for each operation. Your grade will be determined in part by the completeness of your test plan.

Create a batch test driver that executes these test cases. Write the test output to a file, as well as to the screen. Be sure to identify each test case with labels in your output. Print "before" and "after" values to show the results of each operation.

Your test driver may use any of the operations in the Binary Search Tree class from this chapter. Once an operation is tested, you may use it to test other operations.

### ***Turn In***

1. a listing of the package specification with the added operations
2. a listing of the test driver, including the operations to be tested
3. copy of the test plan, with the results of the test cases recorded
7. There is a real program developed by a computer company that reads in written reports, issues warnings on bad style, and partially corrects the style. Your assignment is to create a simplified version of this program. It examines a report for the use of annoying words, cautions the author on the tendencies to use such words, and then corrects the report. After the corrected report is printed, the program user may issue commands to add or delete entries from the collection of annoying words.

### ***Input***

1. From file `Bad_Words`: The list of annoying words is in the following format:

`<annoying word> <synonym>, <synonym>, ..., <synonym>.`

The first word on each line is an annoying word, followed by a blank space and a maximum of five synonyms, separated by commas. The last synonym is followed by a period. Each word is a maximum of 15 characters. Sample line:

grungy dirty,soiled,grimy,encrusted.

The entries are not arranged in any particular order.

2. From file `Text_File`: A paragraph of text over several lines, in which the words are separated from one another by blanks, commas, periods, or quotes. There need not be a blank before the word at the beginning of the line.

3. From the user (keyboard): Any number of the following commands in any order, one per line.

*ADD* *<annoying word>* *<synonym>*, *<synonym>*...,*<synonym>*.

Add *<annoying word>* and its synonyms from the collection of annoying words.

*DELETE* *<annoying word>*

Delete *<annoying word>* and its synonyms from the collection of annoying words.

*QUIT*

Stop processing user commands.

### **Output**

1. Alphabetical listing of the collection of annoying words and their synonyms (to file `New_File`).

2. Echo printed original text from `Text_File` (to screen and file `New_File`).

3. The corrected text after the substitutions of synonyms for annoying words (to the screen and `New_File`).

4. List of slight tendencies to use annoying words. Display a heading ("Slight Tendencies to Use Annoying Words"), followed by any annoying word that occurred one to four times in the input text. The list of words should be printed one word per line, alphabetized, to the screen and `New_File`.

5. A list of extreme tendencies to use annoying words. Display a heading ("Extreme Overuse of Annoying Words"), followed by a list of annoying words that occurred five or more times in the input text. The list of words should be printed one word per line, alphabetized to the screen and `New_File`.

6. Issue prompts for commands to screen. Echo print commands issued by program user to `New_File`.

7. After *QUIT* command, reprint the text from `Text_File`, using the current collection of annoying words and synonyms to correct the text (to screen and `New_File`).

### **Processing**

The program begins by reading in the collection of annoying words from file `Bad_Words`. The annoying words should be stored in a binary search tree (`Bad_Tree`). The list of synonyms associated with each annoying word should also be stored. To make things more interesting, note that when the program finds an annoying word in the text, it must replace it by a more acceptable synonym. However, it is nearly as annoying to have the same synonym replace every occurrence of the annoying word. The list of synonyms for each annoying word should be stored in a circular list so that the program cycles through the synonyms for each occurrence of a particular annoying word (see example). After the `Bad_Tree` has been built, display the annoying words alphabetically, with their synonyms, to `NEW_FILE`.

The program then processes the `Text_File` by reading the text and replacing any annoying words with the next synonym associated with it. The modified text is displayed to both the screen and `New_File`.

After the user sees which words the program has, the author may use the DELETE and ADD commands to modify the collection of annoying words and their synonyms. After all modifications have been made, the user issues the QUIT command. The program then reprocesses the `Text_File` by reading the text and replacing with their synonyms any words that are currently considered annoying. The final text and the lists of slight and extreme tendencies are displayed to the screen and `New_File`.

### ***Sample Input***

*File* `Bad_Words`:

```
grungy dirty,soiled,grimy,encrusted.
awesome amazing,incredible.
teeny small,tiny.
```

*File* `Text_File`:

```
The apartment was so grungy it was totally awesome. The
sofa was grungy, the floor was grungy, the fridge was
grungy, and even the grass outside was grungy. When I
think that could have been where I would live this year,
all I could say was, "Totally awesome."
```

*Commands*:

```
ADD fridge refrigerator.
DELETE awesome
QUIT
```

### ***Sample Output (File New\_File)***

Annoying words and synonyms:

```
awesome amazing, incredible
grungy dirty,soiled,grimy,encrusted
teeny small,tiny
```

Text from `Text_File`:

```
The apartment was so grungy it was totally awesome. The
sofa was grungy, the floor was grungy, the fridge was
grungy, and even the grass outside was grungy. When I
think that could have been where I would live this year,
all I could say was, "Totally awesome."
```

Corrected Text:

```
The apartment was so dirty it was totally amazing. The
sofa was soiled, the floor was grimy, the fridge was
encrusted, and even the grass outside was dirty. When I
think that could have been where I would live this year,
all I could say was "Totally incredible."
```



Slight Tendencies to Use Annoying Words:  
awesome

Extreme Overuse of Annoying Words:  
grungy

ADD fridge refrigerator  
DELETE awesome

Final text:

The apartment was so dirty it was totally awesome. The sofa was soiled, the floor was grimy, the refrigerator was encrusted, and even the grass outside was dirty. When I think that could have been where I would live this year, all I could say was, "Totally awesome."

Slight Tendencies to Use Annoying Words:  
fridge

Extreme Overuse of Annoying Words:  
grungy

### ***Turn In***

1. program listing
2. hard copy of file New\_File

*(This programming assignment was developed from an idea by Gael Buckley.)*

## 13— More Trees and Graphs

### GOALS

After reading this chapter, you should be able to

- show how a binary tree can be represented in an array, with implicit positional links between the elements
- define the following terms:  
full binary tree; complete binary tree; complete binary tree; heap
- describe the shape and order properties of a heap and implement a heap in a nonlinked tree representation in an array
- describe a priority queue at the logical level and implement a priority queue as a heap
- compare the implementation of a priority queue using a heap, linked list, and binary search tree
- define the following terms related to graphs:  
directed graph; undirected graph; vertex; edge; path; complete graph; weighted graph; adjacency matrix; adjacency list
- implement a graph using an adjacency matrix to represent the edges
- explain the difference between a depth-first and a breadth-first search and implement these searching strategies using stacks and queues for auxiliary storage
- implement a shortest-paths operation, using a priority queue to access the edge with the minimum weight

So far we have examined several basic classes in depth, discussing their uses and operations, as well as one or more implementations of each. As we have constructed programmer-defined data structures out of the built-in types provided by our high-level language, we have noted variations that adapt them to the needs of different applications. In Chapter 12 we looked at how a tree structure, the binary search tree, facilitates searching data stored in a linked structure. In this chapter we see how other branching structures are used to model a variety of applications.

### A Nonlinked Representation of Binary Trees

Our discussion of the implementation of binary trees has so far been limited to a scheme in which the pointers from parent to children are *explicit* in the data structure. A field was declared in each node for the pointer to the left child and the pointer to the right child.

A binary tree can be stored in an array in such a way that the relationships in the tree are not physically represented by link fields, but are *implicit* in the algorithms that manipulate the tree stored in the array. The code is, of course, much less self-documenting, but we might save memory space because there are no pointers.

Let's take a binary tree and store it in an array in such a way that the parent-child relationships are not lost. We store the tree in the array level by level, left to right. Figure 13-1 illustrates this mapping.

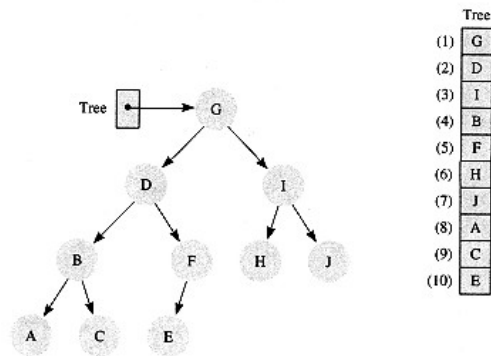


Figure 13-1  
A Binary Tree and Its Array Representation

To implement the algorithms that manipulate the tree, we must find the left and right child of a node in the tree. Comparing the tree and the array in Figure 13-1, we see that

Tree (1) children are in Tree (2) and Tree (3).

Tree (2) children are in Tree (4) and Tree (5).

Tree (3) children are in Tree (6) and Tree (7).

Do you see the pattern? For any node `Tree (Index)`, its left child is in `Tree (Index * 2)` and its right child is in `Tree (Index * 2 + 1)` (provided that these child nodes exist). Notice that the nodes in the array from `Tree (Tree 'Last / 2 + 1)` to `Tree (Tree 'Last)` are leaf nodes.

Not only can we easily calculate the location of a node's children, we also can determine the location of its *parent* node. This task isn't an easy one in a binary tree linked together with pointers from parent to child nodes, but it's very simple in our implicit link implementation:

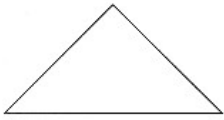
`Tree (Index)'s parent is in Tree (Index / 2).`

Because integer division truncates any remainder, `Index / 2` is the correct parent index for either a left or right child. Thus this implementation of a binary tree is linked in both directions: from parent to child, and from child to parent. We take advantage of this fact later in this chapter.

#### Full Binary Tree

A binary tree in which all of the leaves are on the same level and every nonleaf node has two children.

This tree representation works well for any binary tree that is full or complete. A **full binary tree** is a binary tree in which all of the leaves are on the same level and every nonleaf node has two children. The basic shape of a full binary tree is triangular:



#### Complete Binary Tree

A binary tree that is either full or full through the next-to-last level, with the leaves on the last level as far to the left as possible.

A **complete binary tree** is a binary tree that is either full or full through the next-to-last level, with the leaves on the last level as far to the left as possible. The shape of a complete binary tree is either triangular (if the tree is full) or something like the following:

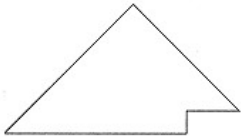


Figure 13-2 shows some examples of binary trees.

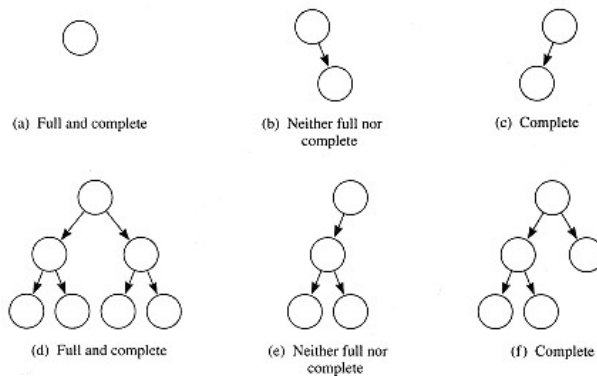


Figure 13-2  
Examples of Binary Trees

The array-based representation is simple to implement for trees that are full or complete, because the elements occupy contiguous array slots. If a tree is not full or complete, however, we must account for the gaps where nodes are missing. To use the array representation, we must store a dummy value in those positions in the array in order to maintain the proper parent-child relationship. The choice of a dummy value depends on what information is stored in the tree. For instance, if the elements in the tree are positive integers, a negative value can be stored in the dummy nodes.

Figure 13-3 illustrates an incomplete tree and its corresponding array. Some of the array slots do not contain actual tree elements, however; they contain dummy values. The algorithms to manipulate the tree must reflect this situation. For example, to determine whether the node in `Tree[Index]` has a left child, you must check whether  $\text{Index} * 2 \leq \text{Tree}.\text{Last}$ , and then check to see if the value in `Tree[Index * 2]` is a dummy value.

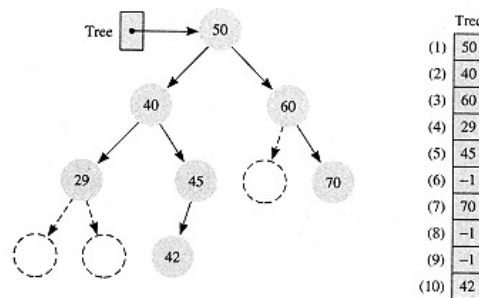


Figure 13-3  
A Binary Search Tree Stored in an Array with Dummy Values

We have seen how an array can be used to represent a binary tree. We can also reverse this process, creating a binary tree from the elements in an array. In fact, we can regard *any* one-dimensional array as representing the nodes in a tree, but the data values that happen to be stored in it may not match this structure in a meaningful way.

## Heaps

### Heap

A complete binary tree, each of whose elements contains a value that is greater than or equal to the value of each of its children.

A **heap** is a binary tree that satisfies two properties: one concerns its shape and the other concerns the order of its elements. The *shape property* is simple: a heap must be a complete binary tree. The *order property* says that, for every node in the heap, the value stored in that node is greater than or equal to the value in each of its children.

Figure 13-4 shows two heaps that contain the values A through J. Notice that the placement of the values differs in the two trees, but the shape is the same: a complete binary tree of 10 elements. Note also that the two heaps have the same root node. A group of values can be stored in a binary tree in many ways and still satisfy the order property of heaps. Because of the shape property, we know that the shape of all the heaps with a given number of elements is the same. Because of the order property, we also know that the root node always contains the largest value in the heap. This fact gives us a hint as to what this

data structure might be good for. The special feature of heaps is that we always know where the maximum value is—it's in the root node.

*Note:* When we refer to a "heap" in this section, we are referring to the structure defined above. This might also be called a "maximum heap" because the root node contains the maximum value in the structure. It's also possible to create a "minimum heap," each of whose elements contains a value that is *less* than or equal to the value of each of its children.

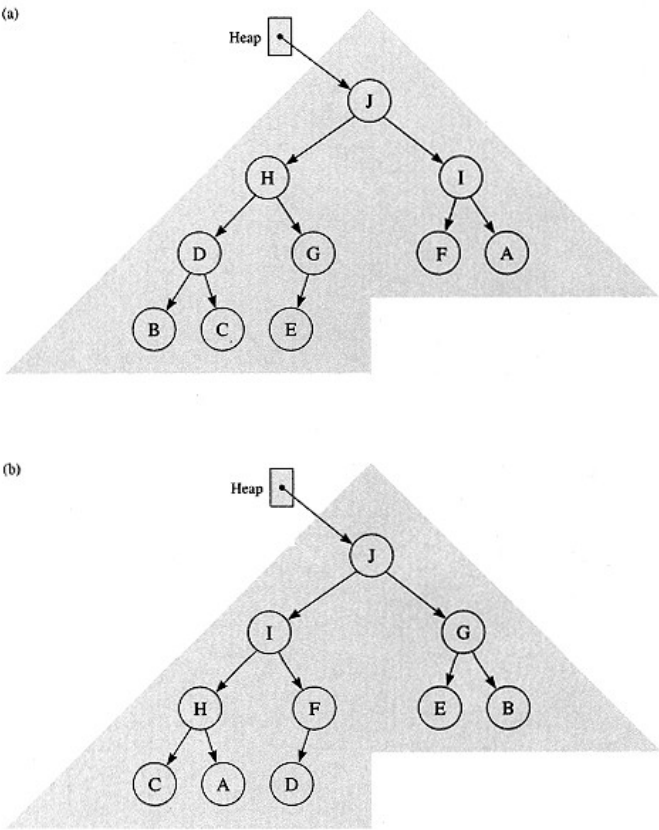


Figure 13-4  
Two Heaps Containing the Letters A–J

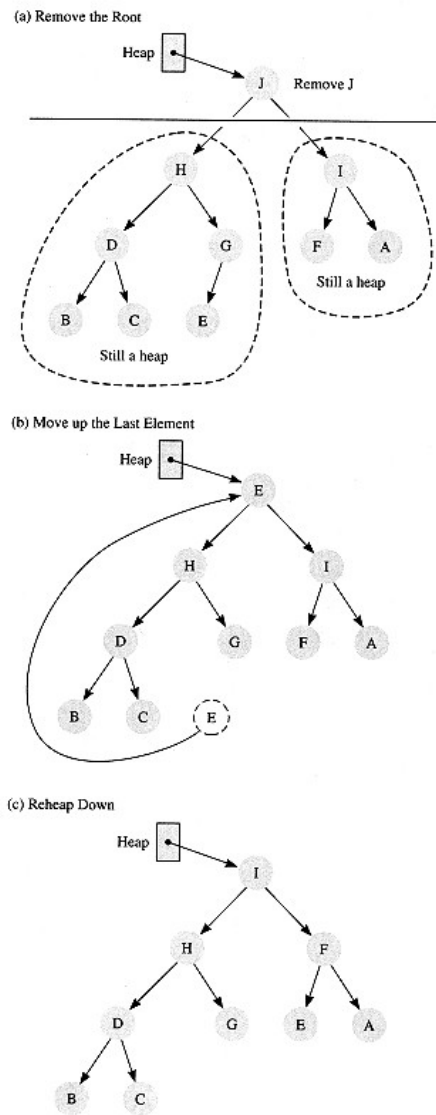


Figure 13-5  
Removing the Root from a Heap

Let's say that we want to remove the element with the largest value from a heap. The largest element is in the root node, so we can easily remove it, as illustrated in Figure 13-5(a). But this leaves a hole in the root position. Because the heap's tree must be complete, we decide to fill the hole with the bottom rightmost element from the heap. Now the structure satisfies the shape property [Figure 13-5(b)]. However, the replacement value came from the bottom of the tree where the smaller values are, so the tree no longer satisfies the order property of heaps.

This situation suggests one of the basic heap operations: given a complete binary tree whose elements satisfy the heap order property *except in the root position*, repair the structure so that it is again a heap. This operation, called *Reheap Down*, involves moving the element down from the root position until it ends up in a position where the order property is satisfied [see Figure 13-5(c)].

Now let's say that we want to add an element to the heap—where do we put it? The shape property tells us that the tree must be complete, so we put the new element in the next bottom rightmost place in the tree, as illustrated in Figure 13-6(a). Now the shape property is satisfied, but the order property may be violated. This situation illustrates the need for another basic heap operation: given a complete binary tree containing  $N$  elements, whose first  $N - 1$  elements satisfy the order property of heaps, repair the structure so that it is again a heap. To fix this structure, we need to float the  $N$ th element up in the tree until it's in its correct place [see Figure 13-6(b)]. This operation is called *Reheap Up*.

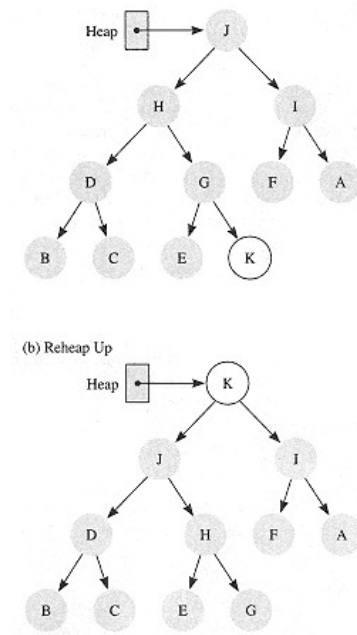


Figure 13-6  
Adding an Element to a Heap



Although we have graphically depicted heaps as binary trees with nodes and links, it would be very impractical to implement the heap operations using the usual linked tree representation. The shape property of heaps tells us that the binary tree is complete, so we know that it never has any holes in it. Thus we can easily store the tree in an array with implicit links. Figure 13-7 shows how the values in a heap would be stored in this array representation. If a heap with  $N$  elements is implemented this way, the shape property says that the heap elements are stored in  $N$  consecutive slots in the array, with the root element in the first slot and the last leaf node in the  $N$ th slot.

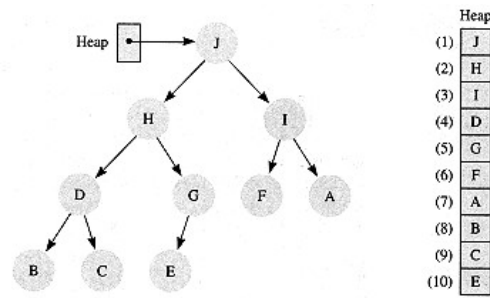


Figure 13-7  
Heap Values in an Array Representation

### A Heap Package Specification

Specification 13-1 contains the specifications of the two heap restoration operations we discussed in the previous section. Most of the generic formal parameters of this package serve the same function as those in our generic List and Tree packages. But unlike these two packages, we have not encapsulated the declaration of a Heap type in our package. Instead, we have made the type for the Heap a generic formal array parameter. The user of this package is responsible for declaring an array type for `Heap_Array`. Because the details of the heap array type are declared by the user, our Heap package is neither an ADT or an ADO. It's simply a service package that supplies two operations for unconstrained one-dimensional arrays. Our heap package contains no insert, delete, traverse, or other operations common with Lists and Trees. Heaps are seldom used alone. Like arrays, heaps are used as building blocks for higher level classes. They are most commonly used to implement priority queues (discussed later in this chapter) and efficient sorting procedures (Chapter 14).

#### Specification 13-1— Heap

```
generic
 type Element_Type is private;
 type Key_Type is limited private;
 type Heap_Array is array (Positive range <>) of Element_Type;
 with function Key_Of (Element : in Element_Type) return Key_Type;
 with function ">" (Left : in Key_Type; Right : in Key_Type) return Boolean;
```

```

package Heap is

-- This package implements a Heap, a binary tree that satisfies two properties:
-- 1. Shape Property : A heap must be a complete binary tree.
-- 2. Order Property : For every node in the heap, the key is greater
-- than or equal to the keys of its children. This
-- order is based on the formal generic ">" function.

-- The keys of the elements in a heap need not be unique.

procedure Reheap_Down (Heap : in out Heap_Array);
-- Restores the order property of heaps to the tree stored in Heap.

-- Preconditions: The order property of heaps is violated only
-- by the first element of Heap.

-- Postconditions: The order property applies to all elements of Heap.

procedure Reheap_Up (Heap : in out Heap_Array);
-- Restores the order property of heaps to the tree stored in Heap.

-- Preconditions: The order property of heaps is violated only
-- by the last element of Heap.

-- Postconditions: The order property applies to all elements of Heap.

end Heap;

```

### ***A Heap Package Body***

The order property says that the key of any parent node in the heap is greater than or equal to the keys of its children. This means that for every *nonleaf* node `Heap (Index)`,

```
Key_Of (Heap(Index)) >= Key_Of (Heap(Index * 2))
```

and, if there is a right child,

```
Key_Of (Heap(Index)) >= Key_Of (Heap(Index * 2 + 1))
```

We've specified two utility operations to fix heaps that violate this order property *at one end or the other*. Now let's look at these operations in more detail.

When the `Reheap_Down` procedure is called, there are two possibilities. If the value of the root node, `Heap (1)`, is greater than or equal to the values of its children, the order property is still intact and we don't have to do anything. Otherwise, we know that the

maximum value of the tree is in either the root node's left child,  $\text{Heap}(2)$ , or the right child,  $\text{Heap}(3)$ . One of these values must be swapped with the smaller value in the root. Now the subtree rooted at the node that was swapped is a heap—except (possibly) for *its* root node. We apply the same process again, asking whether the value in this node is greater than or equal to the values in its children. We test smaller and smaller subtrees of the original heap, moving our original root node down until

1. the root of the current subtree is a leaf node; or
2. the value in the root of the current subtree is greater than or equal to the values of both its children.

The algorithm for this procedure is given below and illustrated with an example in Figure 13-8. With each call,  $\text{Heap}'\text{First}$  is the index of the *root* node that (possibly) violates the heap order property.

### Reheap\_Down

```

if Heap(Root) is not a leaf node then
 -- Does the root have a child with a larger key?
 Max_Child := index of child with larger key
 if Key_Of(Heap(Max_Child)) > Key_Of(Heap(Root)) then
 Swap the values in the root and larger child -- General case
 Reheap_Down(Heap => Heap with root at Max_Child)
 else
 do nothing -- Base case, order intact
 end if
else
 do nothing -- Base case, a leaf node
end if

```

This is a recursive algorithm. In the general case, we swap the values in the root node with its largest child, and then repeat the process. On the recursive call, we pass the subtree whose root is  $\text{Max\_Child}$ ; this shrinks the size of the tree still to be processed, satisfying the smaller-caller question. There are two base cases: (1) if  $\text{Heap}(\text{Root})$  is a leaf, and (2) if the heap order property is already intact. In either of these cases, we do nothing.

How do we determine whether the root node is a leaf? We make use of another property of our mapping of a complete tree into an array (Figure 13-7). Elements with indices between  $\text{Heap}'\text{First}$  and  $\text{Heap}'\text{Last} / 2$  have children. Elements with indices between  $\text{Heap}'\text{Last} / 2 + 1$  and  $\text{Heap}'\text{Last}$  are leaves. So if the index of our (current) root is less than or equal to  $\text{Heap}'\text{Last} / 2$ , we know it's not a leaf.

To determine  $\text{Max\_Child}$  (the index of the child with the larger key), we first check to see whether the current root node has only a single child. If so, it's a left child (because the tree is complete), and we set  $\text{Max\_Child}$  to its index. Otherwise we compare the values of the two child nodes and set  $\text{Max\_Child}$  to the index of the node that has the larger value. The complete procedure  $\text{Reheap\_Down}$  is given in Body 13-1.

The converse operation,  $\text{Reheap\_Up}$ , takes a leaf node that violates the order property of heaps and moves it up until its correct position is found. We compare the value of the bottom node with the value of its parent node. If the parent's value is smaller, the order property is violated so the two nodes are swapped. Then we examine the parent, repeating the process until

1. the current node is the root of the heap; or
2. the value in the current node is less than or equal to the value of its parent node.

The algorithm for this procedure is given below and illustrated in Figure 13-9.

### Reheap\_Up

```

if Heap(Bottom) is not the root node then
 Parent := Index of parent of Heap(Bottom)
 if Key_Of (Heap(Bottom)) > Key_Of (Heap(Parent)) then
 Swap Heap(Bottom) and Heap(Parent) -- General case
 Reheap_Up (Heap => Heap with Parent as new Bottom)
 else
 do nothing -- Base case, order intact
 end if;
else
 do nothing -- Base case, root node
end if

```

This is also a recursive algorithm. In the general case, we swap the (current) "bottom" node with its parent and reinvoke the procedure. On the recursive call, we specify Parent as the bottom node; this shrinks the size of the tree still to be processed, so the smaller-caller question can be answered affirmatively. There are two base cases: (1) if we have reached the root node, or (2) if the heap order property is satisfied. In either of these cases, we exit the procedure without doing anything.

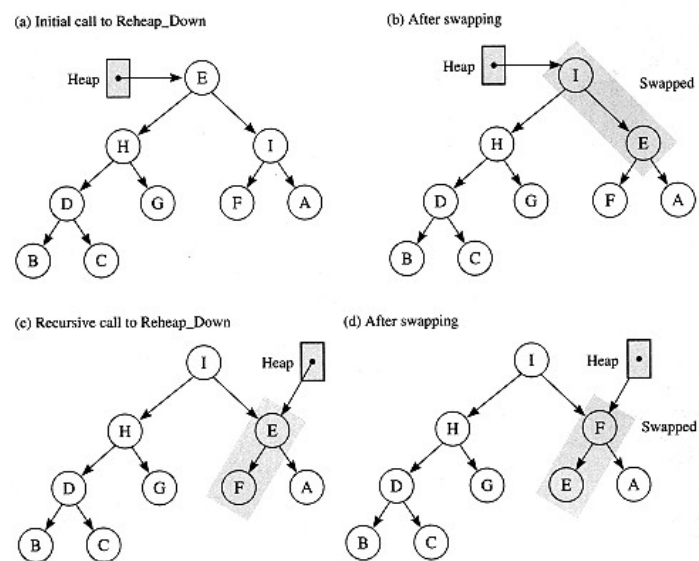


Figure 13-8  
The Reheap\_Down Operation

It is easy to determine whether or not the current node is the root node: the index of the root node is `Heap.First`. How do we find the parent node so that we can test the heap order property? This task isn't easy in a binary tree linked together with pointers from parent to child nodes, but, as we saw earlier, it's very simple in our implicit link implementation:

Index of Parent := Index of Bottom / 2;

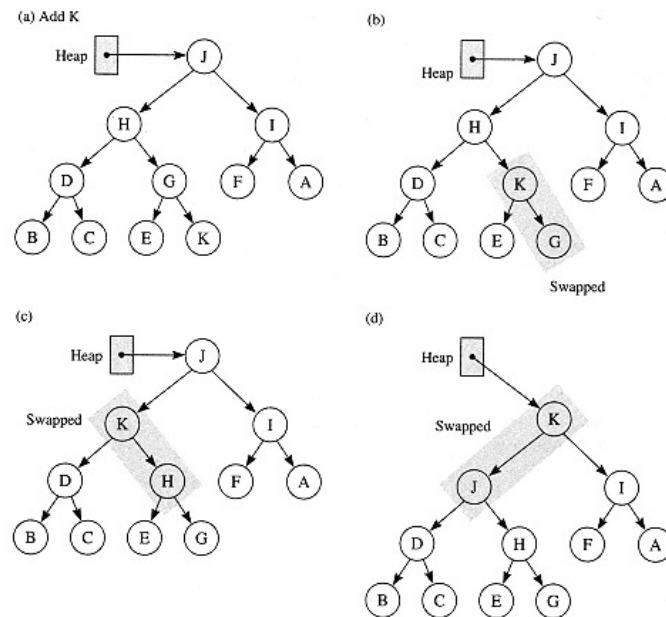


Figure 13-9  
The Reheap\_Up Operation

Body 13-1 contains the code for `Reheap_Up`. As we mentioned earlier, heaps are rarely used alone. Like arrays, they're used as building blocks for higher level classes. We look at one such class in the next section.

### Body 13-1— Heap Implements Specification 13-1

package body Heap is

```

procedure Swap (Left : in out Element_Type; Right : in out Element_Type) is
-- Swaps the values of two variables
 Temp : Element_Type;
```

```

begin
 Temp := Left;
 Left := Right;
 Right := Temp;
end Swap;

function Larger_Child (Heap : in Heap_Array) return Positive is
-- This function returns the index of the root node's child with the larger key.

-- Precondition: The root node has at least a left child.

 Left_Child : Positive; -- Index of the left child node
 Right_Child : Positive; -- Index of the right child node
begin
 Left_Child := Heap'First * 2;
 Right_Child := Heap'First * 2 + 1;
 -- Does the root node have a right child?
 if Right_Child > Heap'Last then
 return Left_Child; -- There is no right child, left child is "larger." Pick it.
 else -- Find the greater of the two children.
 if Key_Of (Heap(Left_Child)) > Key_Of (Heap(Right_Child)) then
 return Left_Child;
 else
 return Right_Child;
 end if;
 end if;
end Larger_Child;

procedure Reheap_Down (Heap : in out Heap_Array) is
 Root : Positive; -- Index of the root node
 Max_Child : Positive; -- Index of child with larger key
begin
 Root := Heap'First;
 -- Check for Base Case 1: Heap(Root) is a leaf
 if Root <= Heap'Last / 2 then
 -- Root is not a leaf so find the child with the larger key
 Max_Child := Larger_Child (Heap);
 -- Check for Base Case 2: Order property intact
 if Key_Of (Heap(Max_Child)) > Key_Of (Heap(Root)) then
 -- General Case: Swap and reheap
 Swap (Heap(Root), Heap(Max_Child));
 Reheap_Down (Heap => Heap(Max_Child..Heap'Last));
 end if;
 end if;
end Reheap_Down;

procedure Reheap_Up (Heap : in out Heap_Array) is
 Bottom : Positive; -- Index of the bottom node in the heap
 Parent : Positive; -- Index of the parent of Bottom

```

```

begin -- Reheap_Up
 Bottom := Heap'Last;
 -- Check for Base Case 1: Heap(Bottom) is the root
 if Bottom > Heap'First then
 -- Bottom is not the root. Determine its parent
 Parent := Bottom / 2;
 -- Check for Base Case 2: Order property intact
 if Key_Of (Heap(Bottom)) > Key_Of (Heap(Parent)) then
 -- General Case: Swap and reheap
 Swap (Heap(Parent), Heap(Bottom));
 Reheap_Up (Heap => Heap(Heap'First..Parent));
 end if;
 end if;
end Reheap_Up;

end Heap;

```

## Priority Queues

In discussing FIFO queue applications in Chapter 7, we said that the operating system of a multiuser computer system may use job queues to save users' requests in the order in which they are made. Another way such requests may be handled is according to how important the job request is. That is, the head of the company might get higher priority than the lowly junior programmer. Or an interactive program might get higher priority than a batch job to print out a report that isn't needed until the next day. To handle these requests efficiently, the operating system may use a structure called a *priority queue*. Because a heap gives us fast access to the largest (or highest-priority) element in the structure, it's a good way to implement a priority queue.

### *The Abstract Level*

A priority queue is an abstract data type with an interesting accessing function: only the *highest-priority* element can be accessed. "Highest priority" can mean different things, depending on the application. Consider, for example, a small company with one secretary. When the other employees leave work on the secretary's desk, which jobs get done first? The jobs are processed in order of the employee's importance in the company; the secretary completes the president's work before starting the vice-president's, and does the marketing director's work before the staff programmer's. That is, the *priority* of each job relates to the level of the employee who initiated it.

In a telephone answering system, calls are answered in the order that they are received. That is, the highest-priority call is the one that has been waiting the longest. Thus a FIFO queue can be considered a priority queue whose highest-priority element is the one that has been queued the longest time.

The operations defined for the Priority Queue include procedures to enqueue items, dequeue items, and clear a priority queue, as well as functions to test for an empty or full priority queue. These operations are very similar to those specified for the FIFO queue discussed in Chapter 7. The Enqueue operation adds a given element to the priority queue. The Dequeue operation removes the highest-priority element from the priority queue and returns it to the user. Specification 13-2 describes these operations.

## Specification 13-2— Priority Queue

generic

```
type Element_Type is private;
type Priority_Type is limited private;
with function Priority_Of (Element : in Element_Type) return Priority_Type;
with function ">" (Left : in Priority_Type; Right : in Priority_Type) return Boolean;
```

package Priority\_Queue is

```
-- This package implements a Priority Queue.
-- The priorities are ordered by the generic formal function ">".
```

```
type Queue_Type (Max_Size : Positive) is limited private; -- The queue class
```

```
OVERFLOW : exception;
UNDERFLOW : exception;
```

```

procedure Clear (Queue : in out Queue_Type);
-- This procedure sets the queue to empty.
--
-- Preconditions: None
--
-- Postconditions: Queue is empty
```

```

procedure Enqueue (Queue : in out Queue_Type;
 Item : in Element_Type);
-- This procedure adds Item to the queue.
--
-- Preconditions: None
--
-- Postconditions: Queue = original Queue with Item included
--
-- Exceptions: OVERFLOW is raised if there is no room in the queue
-- for the Item. The queue is not changed.
```

```

procedure Dequeue (Queue : in out Queue_Type;
 Item : out Element_Type);
-- This procedure removes and returns the element with the
-- highest priority in the queue.
--
-- Preconditions: None
--
-- Postconditions: Queue = original Queue with highest priority element removed
-- Item = highest priority element of original queue
--
-- Exceptions: UNDERFLOW is raised if the queue is empty.
-- The queue is not changed.
```



```

function Full (Queue : in Queue_Type) return Boolean;
-- This function determines whether the queue is full or not.
--
-- Preconditions: None
--
-- Postconditions: Full = (No room in Queue for additional elements)

function Empty (Queue : in Queue_Type) return Boolean;
-- This function determines whether the queue is empty or not.
--
-- Preconditions: None
--
-- Postconditions: Empty = (No elements in Queue)

private

 -- We fill this in later.

end Priority_Queue;

```

### ***The Application Level: Work Request Management***

Let's look in a little more detail at our example of a small company with a single secretary. The secretary would like a program to help manage the work requests that other employees submit. These requests are prioritized in the following order:

|                 |                  |
|-----------------|------------------|
| President       | Highest Priority |
| Vice Presidents |                  |
| Managers        |                  |
| Staff           | Lowest Priority  |

We can use an enumeration type to represent these priorities.

```
type Title_Type is (Staff, Manager, Vice_President, President);
```

Notice that we have arranged these job titles from lowest to highest priority. A work request can be modeled as a record.

```

type work_Request_Rec is
 record
 Name : Name_String; -- Name of person submitting the request
 Title : Title_Type; -- Title of person submitting the request
 Work : Description_Type; -- Description of the work
 end record;

```

We have now defined the type of the element (`Work_Request_Rec`) for our priority queue and the type on which priorities are based (`Title_Type`). Our generic priority

queue package also requires that we supply two functions. The first function returns the priority of an element. The priority of a work request is determined by the job title of the requestor. Here is a function that returns that information:

```
function Title_Of (Request : Work_Request_Rec) return Title_Type is
begin
 return Request.Title;
end Title_Of;
```

The second function needed to instantiate a priority queue of work requests is one to determine whether one priority value is greater than another. In our application, a priority value is an enumeration type (`Title_Type`). Our declaration of `Title_Type` gives us all the relational operators for this type so we don't have to write the comparison function ourselves. Here then is the code to instantiate a priority queue ADT with work requests as queue elements and priority by job title:

```
-- Job queue giving the President highest priority
package Job_Queue is new Priority_Queue (Element_Type => Work_Request_Rec,
 Priority_Type => Title_Type,
 Priority_Of => Title_Of,
 ">" => ">");
```

Now suppose someone in the company discovers that more profits are made when the staff's job requests are completed before the president's requests. That is, to maximize profits, we need to reverse our current priorities. We could make this change in our program by reversing the order of the values in the enumeration type `Title_Type`. But what if that original order is required by other parts of our program? Reversing the declaration order might require us to make many changes elsewhere in the program. There is another way to reverse our priorities: supply the "reverse" relational operator in the instantiation. Then our instantiation becomes

```
-- Job queue giving the President lowest priority
package Job_Queue is new Priority_Queue (Element_Type => Work_Request_Rec,
 Priority_Type => Title_Type,
 Priority_Of => Title_Of,
 ">" => "<");
```

This instantiation creates a priority queue whose highest-priority element is the one with the lowest value of job title.

### ***The Implementation Level***

There are many ways to implement a priority queue. In any implementation, we want to be able to access the largest element quickly and easily. For instance, we can keep the elements in a linked list ordered from largest (highest-priority) element to smallest (lowest-priority) element, or in a binary search tree. We discuss these implementations more at the end of this section.

A heap is an excellent way to implement a priority queue. Here are the appropriate private part declarations for Specification 13-2 to implement a heap-based priority queue:

```

private

-- An unconstrained array of priority queue elements
type Queue_Array is array (Positive range <>) of Element_Type;
type Queue_Type (Max_Size : Positive) is
 record
 Count : Natural := 0;
 Items : Queue_Array (1..Max_Size);
 end record;

end Priority_Queue;

```

We keep the number of elements in a priority queue in `Count`. Using the heap implementation described earlier, the elements are stored in the first `Count` slots of array `Items`. Because of the order property, we know that the largest element is in the root—that is, in the first array slot.

Let's look first at the Dequeue operation. The root element is returned to the caller. After we remove the root, we are left with two subtrees, each of which satisfies the heap property (review Figure 13-5 on page 600). Of course, we can't leave a hole in the root position because that violates the shape property. As we have removed an element, there are now `Count - 1` elements left in the priority queue, stored in array slots 2 through `Count`. If we fill the hole in the root position with the bottom element, array slots 1 through `Count - 1` contain the heap elements. The heap shape property is now intact, but the order property may be violated. The resulting structure is not a heap, but it is almost a heap—all of the nodes *except the root node* satisfy the order property. This is an easy problem to correct, for we have a heap operation to do exactly this task: `Reheap_Down`. Here is our algorithm for Dequeue:

### Dequeue

```

Item := root element of heap
Move last leaf element into the root position
Decrement Count
Reheap_Down

```

The Enqueue operation involves adding an element in its "appropriate" place in the heap. Where is this place? If the new element's priority is larger than the current root element's priority, we know that the new element belongs in the root. But that's not the typical case, so we want a more general solution. To start, we can put the new element at the bottom of the heap, in the next available leaf position (review Figure 13-6 on page 601). Now the array contains elements in the first `Count + 1` slots, preserving the heap shape property. The resulting structure is probably not a heap, but it's *almost* a heap—the order property is violated in the last leaf position. This problem is easy to solve, using the `Reheap_Up` operation. Here is our algorithm for Enqueue:

### Enqueue

```

Count := Count + 1
Put Item in next available leaf position
Reheap_Up

```

Body 13-2 gives the Ada code for all the priority queue operations.

**Body 13-2—****Priority Queue Implements Specification 13-2**

```

with Heap;
package body Priority_Queue is

 -- Instantiate a heap service package using the declarations from the
 -- priority queue's generic formal parameters and private part.

 package Queue_Heap is new Heap (Element_Type => Element_Type,
 Key_Type => Priority_Type,
 Heap_Array => Queue_Array,
 Key_Of => Priority_Of,
 ">" => ">");

 procedure Enqueue (Queue : in out Queue_Type;
 Item : in Element_Type) is
 begin
 if Queue.Count = Queue.Max_Size then
 raise OVERFLOW;
 else
 -- Put Item in the next free array slot
 Queue.Count := Queue.Count + 1;
 Queue.Items(Queue.Count) := Item;
 -- At this point, the heap order property applies to Items(1..Count-1).
 -- It is violated only by the last leaf node. Fix it.
 Queue_Heap.Reheap_Up (Heap => Queue.Items(1..Queue.Count));
 end if;
 end Enqueue;

 procedure Dequeue (Queue : in out Queue_Type;
 Item : out Element_Type) is
 begin
 if Queue.Count = 0 then
 raise UNDERFLOW;
 else
 Item := Queue.Items(1); -- Return the root element
 -- Move the last leaf element into the root position
 Queue.Items(1) := Queue.Items(Queue.Count);
 Queue.Count := Queue.Count - 1;
 -- At this point the heap order property applies to Items(2..Count).
 -- It is violated only by the root node. Fix it.
 Queue_Heap.Reheap_Down (Heap => Queue.Items(1..Queue.Count));
 end if;
 end Dequeue;

```

```

function full (Queue : in Queue_Type) return Boolean is
begin
 return Queue.Count = Queue.Max_Size;
end Full;

function Empty (Queue : in Queue_Type) return Boolean is
begin
 return Queue.Count = 0;
end Empty;

procedure Clear (Queue : in out Queue_Type) is
begin
 Queue.Count := 0;
end Clear;

end Priority_Queue;

```

### ***Heaps Versus Other Representations of Priority Queues***

How efficient is the heap implementation of a priority queue? The clear, full, and empty operations are trivial, so we only examine the operations to add and remove elements. `Enqueue` puts the new element into the next free leaf node in the tree. This array position can be accessed directly, so this part of the operation is  $O(1)$ . `Reheap_Up` is then invoked to correct the order. This operation moves the new element up the tree, level by level; because a complete tree is of minimum height, there are at most  $\log_2 N$  levels above the new element ( $N = \text{Count}$ ). So `Enqueue` is an  $O(\log_2 N)$  operation. `Dequeue` removes the element in the root node and replaces it with the bottom rightmost leaf node. Both of these elements in the array can be accessed directly, so this part of the operation is  $O(1)$ . Then `Reheap_Down` is invoked to correct the order. This operation moves the root element down in the tree, level by level. There are at most  $\log_2 N$  levels below the root, therefore `Dequeue` is also an  $O(\log_2 N)$  operation.

How does this implementation compare to the others we mentioned earlier in this section? If we implement the priority queue with a linked list, ordered from largest to smallest element, `Dequeue` merely removes the first node from the list—an  $O(1)$  operation. `Enqueue`, however, must search up to all the elements in the list to find the appropriate insertion place; thus it's an  $O(N)$  operation.

If the priority queue is implemented using a binary search tree, the efficiency of the operations depends on the shape of the tree. When the tree is bushy, both `Enqueue` and `Dequeue` are  $O(\log_2 N)$  operations. In the worst case, if the tree degenerates to a linked list, ordered from smallest to largest element, both `Enqueue` and `Dequeue` are  $O(N)$ . The following table summarizes the efficiency of the different implementations:

Comparison of Priority Queue Implementations

|                           | <i>Enqueue</i> | <i>Dequeue</i> |
|---------------------------|----------------|----------------|
| <i>Heap</i>               | $O(\log_2 N)$  | $O(\log_2 N)$  |
| <i>Linked List</i>        | $O(N)$         | $O(1)$         |
| <i>Binary Search Tree</i> |                |                |
| Bushy                     | $O(\log_2 N)$  | $O(\log_2 N)$  |
| Skewed                    | $O(N)$         | $O(N)$         |

Overall, the binary search tree looks good if it's balanced. However, a binary search tree can become skewed, which reduces the efficiency of the operations. Another disadvantage of the binary search tree is that it may not contain duplicate priorities (keys). This restriction makes the binary tree implementation unsuitable for the work request application we discussed earlier because many different employees have the same job title (priority). The heap, on the other hand, is always a tree of minimum height. The heap isn't a good structure for accessing any randomly selected element, but that's not one of the operations defined for priority queues. The accessing function of a priority queue specifies that only the largest (or highest-priority) element can be accessed. The linked list is excellent for this operation (assuming the list is ordered from largest to smallest), but we may have to search the whole list to find the place to add a new element. Therefore, for the operations specified for priority queues the heap is an excellent choice.

Graphs

The Abstract Level

Binary trees provide a very useful way of representing relationships in which a hierarchy exists. That is, a node is pointed to by at most one other node (its parent), and each node points to at most two other nodes (its children). If we remove the restriction that each node can have at most two children, we have a general tree, as pictured in Figure 13-10.

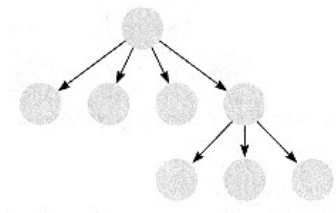


Figure 13-10  
A General Tree

---

|                                                                                                          |
|----------------------------------------------------------------------------------------------------------|
| <b>Graph</b>                                                                                             |
| A data structure that consists of a set of nodes and a set of edges that relate the nodes to each other. |
| <b>Vertex</b>                                                                                            |
| A node in a graph.                                                                                       |
| <b>Edge</b>                                                                                              |
| Connection between two nodes in a graph.                                                                 |

---

If we also remove the restriction that each node may only have one parent node, we have a data structure called a **graph**. A graph is made up of a set of nodes called **vertices** and a set of lines called **edges** (or arcs) that connect the nodes.

The set of edges describes relationships between the vertices. For instance, if the vertices are the names of cities, the edges that link the vertices could represent roads between

---

**Undirected Graph**

A graph in which the edges have no direction.

**Directed Graph (Digraph)**A graph in which each edge is directed from one vertex to another (or the same) vertex.

---

pairs of cities. Because the road that runs between Houston and Austin also runs between Austin and Houston, the edges in this graph have no direction. This is called an **undirected graph**. However, if the edges that link the vertices represent flights from one city to another, the direction of each edge *is* important. The existence of a flight (edge) from Houston to Austin does not assure the existence of a flight from Austin to Houston. A graph whose edges are directed from one vertex to another is called a **directed graph**, or **digraph**.

There is a great deal of formal mathematics associated with graphs. In fact, there is an area of mathematics called graph theory. In future computing courses you will probably analyze graphs and prove theorems about them. This textbook introduces the graph as an abstract data type, teaches some basic terminology, discusses how a graph might be implemented, and describes how algorithms that manipulate graphs make use of stacks, queues, and priority queues.

Formally, a *graph* is defined as follows:

$$G = (V, E)$$

where

$V(G)$  is a finite, nonempty set of vertices

$E(G)$  is a set of edges (pairs of vertices)

The set of vertices is specified by listing them in *set notation*, within  $\{ \}$  brackets. The following set defines the four vertices of the graph pictured in Figure 13-11(a):

$$V(\text{Graph1}) = \{A, B, C, D\}$$

The set of edges is specified by listing a sequence of edges. Each *edge* is denoted by writing the names of the two vertices it connects in parentheses, with a comma between them. For instance, the vertices in Graph1 in Figure 13-11(a) are connected by the four edges described below:

$$E(\text{Graph1}) = \{(A,B), (A,D), (B,C), (B,D)\}$$

Because Graph1 is an undirected graph, the order of the vertices in each edge is unimportant. The set of edges in Graph1 can also be described as follows:

$$E(\text{Graph1}) = \{(B,A), (D,A), (C,B), (D,B)\}$$

If the graph is a *digraph*, the direction of the edge is indicated by which vertex is listed first. For instance, in Figure 13-11(b), the edge (5, 7) represents a link from vertex 5 to vertex 7. However, there's no corresponding edge (7, 5) in Graph2. Note that in pictures of digraphs, the arrows indicate the direction of the relationship.

---

**Adjacent Nodes**Two nodes in a graph that are connected by an edge.

---

If two vertices in a graph are connected by an edge, they are said to be **adjacent**. In Graph1 [Figure 13-11(a)], vertices A and B are adjacent, but vertices A and C are not. If the vertices are connected by a directed edge, then the first vertex is said to be *adjacent to* the second, and the second vertex is said to be *adjacent from* the first. For example, in Graph2 [in Figure 13-11(b)], vertex 5 is adjacent to vertices 7 and 9, whereas vertex 1 is adjacent from vertices 3 and 11.

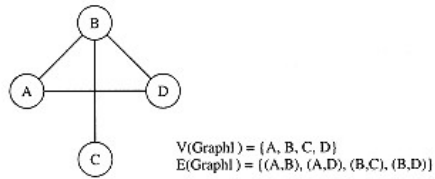
The picture of Graph3 in Figure 13-11(c) may look familiar; it's the tree we looked at earlier in connection with nonlinked representation of a binary tree. A tree is a special case of a directed graph, in which each vertex may only be adjacent from one other vertex (its parent node) and one vertex (the root) is not adjacent from any other vertex.

**Path**

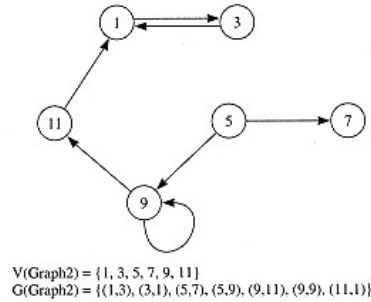
Sequences of vertices that connects two nodes in a graph.

A **path** from one vertex to another consists of a sequence of vertices that connect them. For a path to exist, there must be an uninterrupted sequence of edges from the first vertex, through any number of vertices, to the second vertex. For example, in Graph2, there's a path from vertex 5 to vertex 3, but not from vertex 3 to vertex 5. Note that in a tree, such as Graph3 [Figure 13-11(c)], there's a unique path from the root to every other node in the tree.

(a) Graph1 is an undirected graph.



(b) Graph2 is a directed graph.



(c) Graph3 is a directed graph.

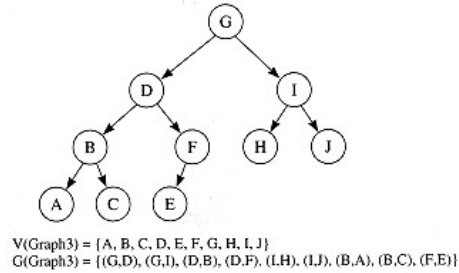


Figure 13-11  
Some Examples of Graphs



**Complete Graph**

A graph in which every vertex is connected to every other vertex.

A **complete graph**\* is one in which every vertex is adjacent to every other vertex. Figure 13-12 shows two complete graphs. If there are  $N$  vertices, there are  $N * (N - 1)$  edges in a complete directed graph and  $N * (N - 1)/2$  edges in a complete undirected graph.

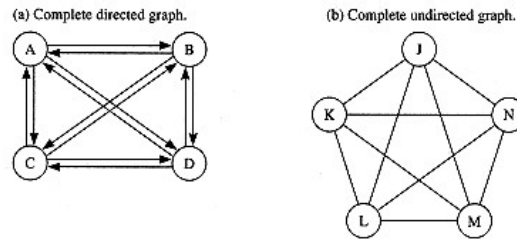


Figure 13-12  
Two Complete Graphs

**Weighted Graph**

A graph in which each edge carries a value.

A **weighted graph** is a graph in which each edge carries a value. Weighted graphs can be used to represent applications in which the *value* of the connection between the vertices is important, not just the *existence* of a connection. For instance, in the weighted graph pictured in Figure 13-13, the vertices represent cities and the edges indicate the Air Busters Airlines flights that connect the cities. The weights attached to the edges represent the air distances between pairs of cities.

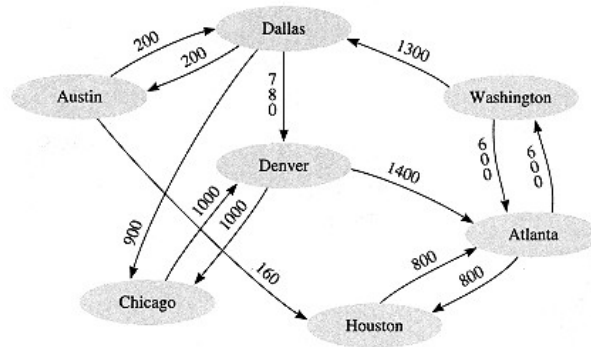


Figure 13-13  
A Weighted Graph

To see whether we can get from Denver to Washington, we look for a path between them. If the total travel distance is determined by the sum of the distances between each pair of cities along the way, we can calculate the travel distance by adding the weights attached to the edges that constitute the path between them. Note that there may be multiple paths between two vertices. Later in this chapter, we talk about a way to find the shortest path between two vertices.

\*Note that the term *complete* means different things when applied to trees and graphs.

We have described a graph at the abstract level as a set of vertices and a set of edges that connect some or all of the vertices one to another. What kind of operations are defined on a graph? In this chapter we specify and implement a small set of useful graph operations. Many other operations on graphs can be defined; we have chosen operations that are useful in the graph applications described later in the chapter.

Specification 13-3 describes our weighted graph. Most of the generic formal parameters of this package are similar to those we used in our list and binary search tree packages. `Vertex_Type` describes the information the user wants to store in every vertex. A key of type `Key_Type` uniquely identifies each vertex in the graph. Because we need to assign values of type `Key_Type` in the package body, we haven't declared it as limited. As a result, we must use the name `Equal` instead of `"="` for the function that compares two keys for equality. Note that in Ada 95, we can use `"="` for a nonlimited generic formal parameter.

### Specification 13-3— Weighted Graph

generic

```

type Vertex_Type is private; -- For information stored in a graph vertex
type Key_Type is private; -- For uniquely identifying a vertex
type Weight_Type is private; -- For information stored with an edge
with function Key_Of (Vertex : in Vertex_Type) return Key_Type;
with function Equal (Left : in Key_Type; Right : in Key_Type) return Boolean;
```

package Graph is

*-- This package implements some of the basic graph operations.*

*-- A graph consists of a set of vertices and a set of weighted  
-- edges that connect some or all of the vertices one to another.*

```

type Graph_Type (Max_Vertices : Positive) is limited private; -- The graph class
```

*-- Additional types for graph operation parameters*

*-- An edge is described by a pair of vertex keys*

```

type Edge_Type is
 record
 From : Key_Type;
 To : Key_Type;
 end Record;
```

*-- An array based list of Keys. Get\_Adjacent\_Vertices returns a list of this type.*

```

type Key_Array is array (Positive range <>) of Key_Type;
type Key_List (Max_Vertices : Positive) is
 record
 Length : Natural := 0;
 Items : Key_Array (1..Max_Vertices);
 end record;
```

```

-- Exceptions
VERTEX_ERROR : exception;
EDGE_ERROR : exception;
OVERFLOW : exception;

procedure Clear (Graph : in out Graph_Type);
-- Makes a graph empty.

-- Preconditions: None

-- Postconditions: V(Graph) = empty
-- E(Graph) = empty

procedure Add_Vertex (Graph : in out Graph_Type;
 Vertex : in Vertex_Type) ;
-- Add Vertex to the Graph

-- Preconditions: None

-- Postconditions: Vertex is an unmarked vertex in Graph.

-- Exceptions: VERTEX_ERROR is raised if a vertex already exists in the graph
-- with the same key of the Vertex being added.
-- OVERFLOW is raised if Graph has no more room
-- for an additional vertex.

procedure Add_Edge (Graph : in out Graph_Type;
 Edge : in Edge_Type;
 Weight : in Weight_Type);
-- Add Edge with specified Weight to Graph

-- Preconditions: None

-- Postconditions: Edge is an edge in E(Graph) with specified Weight.

-- Exceptions: VERTEX_ERROR is raised if one or both of the vertices
-- defining Edge do not exist in the Graph.

procedure Retrieve (Graph : in Graph_Type;
 Key : in Key_Type;
 Vertex : out Vertex_Type);
-- Returns a copy of the Vertex with the given Key

-- Preconditions: None

-- Postconditions: Vertex contains a copy of the Graph vertex with the given Key.

-- Exceptions: VERTEX_ERROR is raised if there is no vertex with the
-- given Key in the Graph.

```

```

function Weight_Of (Graph : in Graph_Type;
 Edge : in Edge_Type) return Weight_Type;
-- Returns the weight associated with Edge

-- Preconditions: None

-- Postconditions: Weight = weight associated with Edge

-- Exceptions: VERTEX_ERROR is raised if one or both of the vertices
-- defining Edge do not exist in the Graph.
-- EDGE_ERROR is raised if Edge does not exist in the Graph.

procedure Get_Adjacent_Vertices (Graph : in Graph_Type;
 Key : in Key_Type;
 Adj_Keys : out Key_List);
-- Returns a list of the keys of all the vertices adjacent
-- to the node with the given Key.

-- Preconditions: None

-- Postconditions: Adj_Keys contains the keys of all the vertices that are
-- adjacent to the vertex with the given Key.

-- Exceptions: VERTEX_ERROR is raised if there is no vertex with the
-- given Key in the Graph.
-- CONSTRAINT_ERROR is raised if the Max_Size of Adj_Keys is
-- less than the number of adjacent nodes.

procedure Clear_All_Marks (Graph : in out Graph_Type);
-- Marks all vertices in the graph as not visited.

-- Preconditions: None

-- Postconditions: All vertices in Graph are marked as not visited.

procedure Mark_Vertex (Graph : in out Graph_Type;
 Key : in Key_Type);
-- Marks the vertex with the given Key as visited.

-- Preconditions: None

-- Postconditions: The vertex with the given Key is marked as visited

-- Exceptions: VERTEX_ERROR is raised if there is no vertex with the
-- given Key in the Graph.

function Marked (Graph : in Graph_Type;
 Key : in Key_Type) return Boolean;

```

```

-- Determines whether the vertex with the given key is marked as visited

-- Preconditions: None

-- Postconditions: Marked = (Vertex with Key has been visited)

-- Exceptions: VERTEX_ERROR is raised if there is no vertex with the
-- given Key in the Graph.

private

-- We fill this in later.

end Graph;
```

From your experience with lists and trees and our general discussions of graphs, the purpose of the constructors `Clear`, `Add_Vertex`, and `Add_Edge` and the inspectors `Retrieve` and `Weight_Of` should already be clear.

The remaining operations have no counterparts in the packages we developed for lists and binary trees. Procedure `Get_Adjacent_Vertices` returns a list of the vertices adjacent to a given vertex. For example, with the graph shown in Figure 13-13, the list of vertex keys returned for Denver consists of Chicago and Atlanta. `Clear_All_Marks`, `Mark_Vertex`, and `Marked` are used by applications that use backtracking techniques to search a graph. Recall how in the maze application of Chapter 11 we marked squares as visited. Many graph applications mark vertices in a similar fashion. We look at some of them in the next section.

### ***The Application Level: Graph Traversals***

The Graph specification given in the last section included only the most basic operations. It didn't include any traversal operations. As you might imagine, there are many different orders in which we can traverse a graph. As a result, we consider the traversal algorithms a graph application rather than an operation. The basic operations given in our specification allow us to implement different traversals *independent* of how the graph itself is actually implemented.

In Chapter 12, we discussed the postorder tree traversal, which goes to the deepest level of the tree and works up. This strategy of going down a branch to its deepest point and moving up is called a *depth-first* strategy. Another systematic way to visit each vertex in a tree is to visit each vertex on level 0 (the root), then each vertex on level 1, then each vertex on level 2, and so on. Visiting each vertex by level in this way is called a *breadth-first* strategy. With graphs, both depth-first and breadth-first strategies are useful. We outline both algorithms within the context of the airline example.

#### **Depth-First Searching**

One question we can answer with the graph in Figure 13-13 is, "Can I get from city X to city Y on my favorite airline?" This is equivalent to asking "Does a path exist in the graph from vertex X to vertex Y?" Using a depth-first strategy, let's develop an operation that finds a path from `Start_Vertex` to `End_Vertex`.

We need a systematic way to keep track of the cities as we investigate them. With a depth-first search, we examine the first vertex that is adjacent from `Start_Vertex`; if

this is `End_Vertex`, the search is over. Otherwise, we examine all the vertices that are adjacent from this vertex. Meanwhile, we need to store the other vertices that are adjacent from `Start_Vertex`. If a path does not exist from the first vertex, we come back and try the second, third, and so on. Because we want to travel as far as we can down one path, backtracking if the `End_Vertex` is not found, a stack is a good structure for storing the vertices. Here is the algorithm we use:

### Depth First Search

```

Push Start_Vertex onto Stack
loop
 Pop Vertex from Stack
 if Vertex /= End_Vertex
 Get List of Vertex's adjacent vertices
 Push all of the adjacent nodes onto the stack
 end if
 exit when Vertex = End_Vertex or Stack is empty
end loop
if Vertex = End_Vertex then
 Display "Path exists"
else
 Display "Path does not exist"
end if

```

Let's apply this algorithm to the sample airline route graph in Figure 13-13. We want to fly from Austin to Washington. We initialize our search by pushing our starting city onto the stack [Figure 13-14(a)]. At the beginning of the loop we pop the current city, Austin, from the stack. The places we can reach directly from Austin are Dallas and Houston; we push both these vertices onto the stack [Figure 13-14(b)]. At the beginning of the second iteration we pop the top vertex from the stack: Houston. Houston is not our destination, so we resume our search from there. There is only one flight out of Houston, to Atlanta; we push Atlanta onto the stack [Figure 13-14(c)]. Again we pop the top vertex from the stack. Atlanta is not our destination, so we continue searching from there. Atlanta has flights to two cities: Houston and Washington.

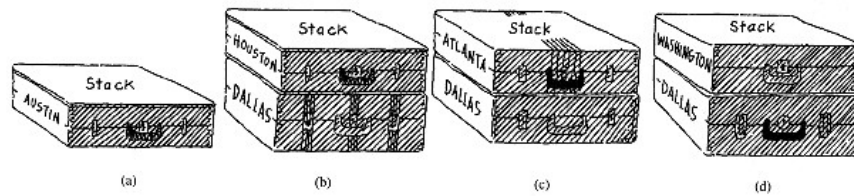


Figure 13-14  
Using a Stack to Store the Routes

But we just came from Houston! We don't want to fly back to cities that we have already visited—this could cause an infinite loop. We have to take care of cycling in this algorithm just as we did in the maze problem in Chapter 11. There we marked a square as having been visited by putting a little stone in the square. Here we must mark a city as having been visited so that it isn't investigated a second time. Let's assume that we have marked the cities that have already been tried, and continue our example. Houston has

already been visited, so we ignore it. The second adjacent vertex, Washington, hasn't been visited so we push it onto the stack [Figure 13-14(d)]. Again we pop the top vertex from the stack. Washington is our destination, so the search is complete. The path from Austin to Washington, using a depth-first search, is illustrated in Figure 13-15.

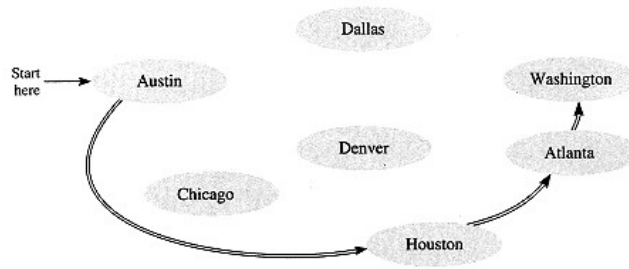


Figure 13-15  
The Depth-First Search

This search is called a depth-first search because we go to the deepest branch, examining all the paths beginning at Houston before we come back to search from Dallas. (The maze problem also used a depth-first search.) When you have to backtrack, you take the branch closest to where you dead-ended. That is, you go as far as you can down one path before you take alternative choices at earlier branches.

Before we look at the source code of the depth-first search operation, let's talk a little more about "marking" vertices on the Graph. Before we begin the search, any marks in the vertices must be cleared to indicate they are not yet visited; this is the purpose of our Graph package's procedure `Clear_All_Marks`. As we visit each vertex during the search, we mark it by calling `Mark_Vertex`. Before we process each vertex we can ask, "Have we visited this vertex before?" The answer to this question is returned by function `Marked`. If we have already visited this vertex, we ignore it and go on.

Now let's look at the Ada code for this search. First, here are the declarations necessary to instantiate a graph package and a stack package for our airline example. Each Vertex in our graph contains the name of the city. The weight of each edge is the number of miles between the two cities.

```

subtype City_String is String (1..10);
subtype Miles_Type is Natural;

function Identity (City : in City_String) return City_String;

package Air_Routes is new Graph (Vertex_Type => City_String,
 Key_Type => City_String,
 Weight_Type => Miles_Type,
 Key_Of => Identity,
 Equal => "=");

package City_Stack is new Stack (Element_Type => City_String);

```

Procedure `Depth_First_Search` is given a graph, a starting city, and an ending city. Because marking the graph changes it, parameter `Graph` must be declared as `in out`. This procedure uses the depth-first strategy to determine if there is a path from the starting city to the ending city, displaying the names of all the cities visited in the search. Note that there is nothing in the procedure listed in Body 13-3 that depends on the implementation of the graph. The procedure is implemented as a graph application; it uses the `Graph` class (including the mark operations) without knowing how the graph is represented.

### Body 13-3—

#### Depth First Search

```

procedure Depth_First_Search (Graph : in out Air_Routes.Graph_Type;
 Start_City : in City_String;
 End_City : in City_String) is
-- Search Graph for path from Start_City to End_City using a depth-first search strategy

 -- Stack stores city names for later LIFO retrieval
 Stack : City_Stack.Stack_Type;

 -- List stores vertices adjacent to the current vertex
 List : Air_Routes.Key_List (Max_Vertices => 10);
 City : City_String; -- Current city

begin
 Air_Routes.Clear_All_Marks (Graph);
 City_Stack.Push (Stack => Stack, New_Element => Start_City);

 City_Loop: -- Each iteration one city (vertex) is processed
 loop
 City_Stack.Pop (Stack => Stack, Popped_Element => City);
 if City /= End_City then
 -- If vertex has not been visited, visit it
 if not Air_Routes.Marked (Graph, City) then
 -- Mark vertex as visited
 Air_Routes.Mark_Vertex (Graph => Graph, Key => City);
 -- Display this city
 Text_IO.Put_Line (City);
 -- Get the vertices adjacent to City
 Air_Routes.Get_Adjacent_Vertices (Graph => Graph,
 Key => City,
 Adj_Keys => List);
 -- Push all adjacent vertices not yet visited
 for Index in 1..List.Length loop
 if not Air_Routes.Marked (Graph, List.Items(Index)) then
 City_Stack.Push (Stack => Stack,
 New_Element => List.Items(Index));
 end if;
 end loop;
 end if; -- if not marked
 else
 -- Display final city

```



```

 Text_IO.Put_Line (City);
end if; -- if not end city
-- Exit when End_City is found or no more cities to examine
exit City_Loop when City = End_City or City_Stack.Empty (Stack);
end loop City_Loop;
end Depth_First_Search;

```

Here is the output produced by this procedure for the graph shown in Figure 13-13, starting at Austin:

```

Austin
Houston
Atlanta
Washington

```

### Breadth-First Searching

A breadth-first search looks at all possible paths at the same depth before it goes to a deeper level. In our flight example, a breadth-first search checks all possible one-stop connections before checking any two-stop connections. For most travelers, this is the preferred approach for booking flights.

When we come to a dead end in a depth-first search, we back up as *little* as possible. We try another route from a recent vertex: the route on top of our stack. In a breadth-first search, we want to back up as *far* as possible to find a route originating from the earliest vertices. The stack is not the right structure for finding an early route. It keeps track of things in the order opposite of their occurrence: the latest route is on top. To keep track of things in the order in which they happened, we use a FIFO queue. The route at the front of the queue is a route from an earlier vertex; the route at the back of the queue is from a later vertex.

To modify the search to use a breadth-first strategy, we change all the calls to stack operations to the analogous FIFO queue operations. Searching for a path from Austin to Washington, we first Enqueue all the cities that can be reached directly from Austin: Dallas and Houston [Figure 13-16(a)]. Then we Dequeue the front queue element. Dallas isn't the destination we seek, so we Enqueue all the adjacent cities that have not yet been visited: Chicago and Denver [Figure 13-16(b)]. (Austin has been visited already, so it isn't Enqueued.) Again we Dequeue the front element from the queue. This element is the other one-stop city, Houston. Houston isn't the desired destination, so we continue the search. There is only one flight out of Houston, and it's to Atlanta. Because we haven't visited Atlanta before, it's Enqueued [Figure 13-16(c)].

Now we know that we can't reach Washington with one stop, so we start examining the two-stop connections. We Dequeue Chicago; this isn't our destination, so we put its adjacent city, Denver, into the queue [Figure 13-16(d)]. Now this is an interesting situation: Denver is in the queue twice. Should we mark a city as having been visited when we put it in the queue or after it has been Dequeued, when we are examining its outgoing flights? If we mark it only after it has been Dequeued, there may be multiple copies of the same vertex in the queue (so we need to check to see if a city is marked *after* it has been Dequeued.)

An alternative approach is to mark the city as having been visited before it's put into the queue. Which is better? It depends on the processing. You may want to know whether there are alternative routes, in which case you would want to put a city into the queue more than once.

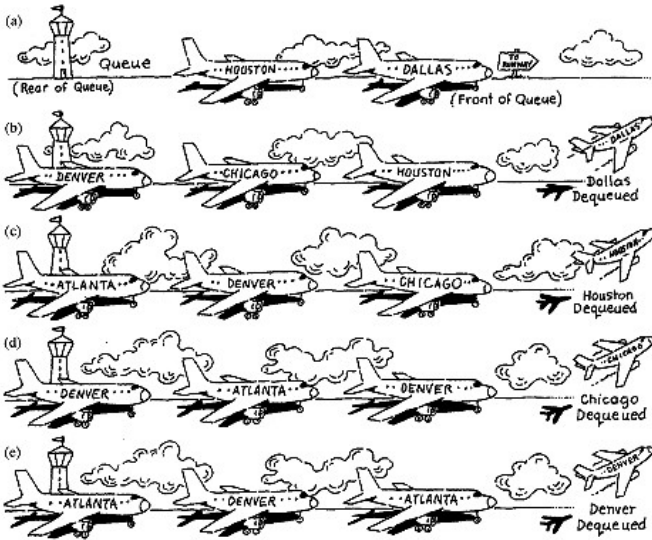


Figure 13-16  
Using a Queue to Store the Routes

Back to our example: we have put Denver into the queue in one step and removed its previous entry at the next step. Denver isn't our destination, so we put its adjacent cities that we haven't already marked (only Atlanta) into the queue [Figure 13-16(e)]. This processing continues until Washington is put into the queue (from Atlanta), and is finally Dequeued. We have found the desired city, and the search is complete. This search is illustrated in Figure 13-17.

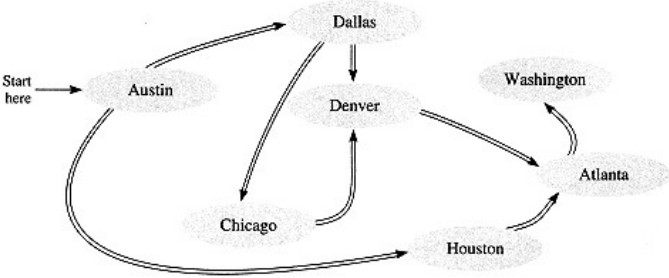


Figure 13-17  
The Breadth-First Search

Body 14-4 gives the source code for the breadth-first search procedure. It's identical to the depth-first search except for the replacement of the stack with a FIFO queue.

#### Body 13-4—

##### Breadth First Search

```

procedure Breadth_First_Search (Graph : in out Air_Routes.Graph_Type;
 Start_City : in City_String;
 End_City : in City_String) is
-- Search Graph for path from Start_City to End_City using a breadth-first search strategy

-- Queue stores city names for later FIFO retrieval
Queue : City_Queue.Queue_Type;

-- List stores vertices adjacent to the current vertex
List : Air_Routes.Key_List (Max_Vertices => 10);
City : City_String; -- Current city

begin
 Air_Routes.Clear_All_Marks (Graph);
 City_Queue.Enqueue (Queue => Queue, Item => Start_City);

 City_Loop: -- Each iteration, one city is examined
 loop
 City_Queue.Dequeue (Queue => Queue, Item => City);
 if City /= End_City then
 -- If vertex has not been visited, visit it
 if not Air_Routes.Marked (Graph, City) then
 -- Mark vertex as visited
 Air_Routes.Mark_Vertex (Graph => Graph, Key => City);
 -- Display this city
 Text_IO.Put_Line (City);
 -- Get the vertices adjacent to City
 Air_Routes.Get_Adjacent_Vertices (Graph => Graph,
 Key => City,
 Adj_Keys => List);
 -- Enqueue all adjacent vertices not yet visited
 for Index in 1..List.Length loop
 if not Air_Routes.Marked (Graph, List.Items(Index)) then
 City_Queue.Enqueue (Queue => Queue,
 Item => List.Items(Index));
 end if;
 end loop;
 end if; -- if not marked
 else
 -- Display final city
 Text_IO.Put_Line (City);
 end if; -- if not end city
 -- Exit when End_City is found or no more cities to examine
 exit City_Loop when City = End_City or City_Queue.Empty (Queue);
 end loop City_Loop;
end Breadth_First_Search;

```

Here is the output produced by this procedure for the graph shown in Figure 13-13, starting at Austin:

Austin  
Dallas  
Houston  
Chicago  
Denver  
Atlanta  
Washington

**The Single-Source Shortest Paths Problem**

We know from the two search operations above that there may be multiple paths from one vertex to another. Suppose that we want to find the shortest path from Austin to each of the other cities that Air Busters serves. By "shortest path" we mean the path whose edge values (weights) added together have the smallest sum. Consider the following two paths from Austin to Washington:

|             |   |            |             |   |            |
|-------------|---|------------|-------------|---|------------|
| Austin      | } | 160 miles  | Austin      | } | 200 miles  |
| Houston     |   |            | Dallas      |   |            |
|             | } | 800 miles  |             | } | 780 miles  |
| Atlanta     |   |            | Denver      |   |            |
|             | } | 600 miles  |             | } | 1400 miles |
| Washington  |   |            | Atlanta     |   |            |
|             |   |            | Washington  |   | 600 miles  |
| <hr/>       |   |            | <hr/>       |   |            |
| Total miles |   | 1560 miles | Total miles |   | 2980 miles |

Clearly the first path is preferable, unless you want to collect frequent-flyer miles.

Let's develop an operation that displays the shortest path from a designated starting city to *every other city* in the graph. This time we aren't searching for a path between a starting city and an ending city. But, as in the two graph searches described earlier, we need an auxiliary structure for storing cities that we process later. By retrieving the city that was most recently put into the structure, the depth-first search tries to keep going "forward." It tries a one-flight solution, then a two-flight solution, then a three-flight solution, and so on. It backtracks to a fewer-flight solution only when it reaches a dead-end. By retrieving the city that had been in the structure the longest time, the breadth-first search tries all one-flight solutions, then all two-flight solutions and so on. The breadth-first search finds a path with a minimum number of flights.

But a minimum *number* of flights does not necessarily mean the minimum total distance. Unlike the depth-first and breadth-first searches, this shortest-path traversal must use the number of miles (edge weights) between cities. But what kind of access helps us retrieve the city in the shortest path? We want to retrieve the city that is *closest to the starting city*. If we consider the flight whose destination is the minimum distance from the start to be the highest-priority element in the structure, then we know of a perfect structure: the priority queue. Our algorithm can use a priority queue whose elements are flights (edges) with their distances *from the starting city*. Here is our declaration for the priority queue element:

```

type Flight_Rec is
 record
 Route : Air_Routes.Edge_Type; -- This flight's start and end cities
 Total : Miles_Type; -- The total miles from starting city
 end record; -- to the destination of this flight

```

Each call to `Dequeue` removes and returns the highest-priority element, the element with the *smallest* value of `Total`.

The following declarations create our priority queue. We get the inverse relationship between priority and distance by associating the formal generic function ">" for comparing keys with the actual generic function "<" for comparing distances.

```

-- This function returns Flight.Total
function Distance_From_Start (Flight : in Flight_Rec) return Miles_Type;

package P_Queue
 is new Priority_Queue (Element_Type => Flight_Rec,
 Priority_Type => Miles_Type,
 Priority_Of => Distance_From_Start,
 ">" => "<");

```

The algorithm for the shortest-path traversal is similar to those we used for the depth-first and breadth-first searches. There are two major differences:

1. We use a priority queue rather than a FIFO queue or stack.
2. We stop only when there are no more cities to process; there is no destination.

The code for the `Shortest_Paths` procedure is given in Body 13-5.

#### Body 13-5— Shortest Paths

```

procedure Shortest_Paths (Graph : in out Air_Routes.Graph_Type;
 Start_City : in City_String) is
-- Determine the shortest paths from Start_City to every other city in Graph

 Flight : Flight_Rec; -- Current flight information
 Current_City : City_String; -- Current city (Destination of Flight)
 Previous_City : City_String; -- The city we came from (Origin of Flight)
 Next_City : City_String; -- A city we go to after Current City
 Distance : Miles_Type; -- Miles from Start_City to Current_City

 -- Priority Queue of Flight_Rec's
 Queue : P_Queue.Queue_Type (Max_Size => 20);

 -- List of adjacent city names
 List : Air_Routes.Key_List (Max_Vertices => 10);

begin
 Air_Routes.Clear_All_Marks (Graph);

```

```

Text_IO.Put_Line (" Shortest Path from " & Start_City);
Text_IO.New_Line;
Text_IO.Put_Line ("Last City Destination Distance");
Text_IO.Put_Line ("-----");

-- Initialize priority queue with starting city
Flight.Route := (From => Start_City, To => Start_City);
Flight.Total := 0;
P_Queue.Enqueue (Queue => Queue, Item => Flight);

City_Loop: -- Each iteration, one city is processed
loop
 -- Exit when all cities have been visited
 exit City_Loop when P_Queue.Empty (Queue);

 -- Get the flight whose destination is closest to Start_City
 P_Queue.Dequeue (Queue => Queue, Item => Flight);
 Current_City := Flight.Route.To;
 Previous_City := Flight.Route.From;
 Distance := Flight.Total; -- Miles traveled from Start_City

 -- If the city hasn't been visited, visit it
 if not Air_Routes.Marked (Graph, Current_City) then
 -- Mark the city as visited
 Air_Routes.Mark_Vertex (Graph => Graph,
 Key => Current_City);

 -- Display the current flight cities and accumulated miles
 Text_IO.Put (Previous_City & " ");
 Text_IO.Put (Current_City & " ");
 Integer_IO.Put (Item => Distance, Width => 6);
 Text_IO.New_Line;

 -- Get list of cities adjacent to the current city
 Air_Routes.Get_Adjacent_Vertices (Graph => Graph,
 Key => Current_City,
 Adj_Keys => List);

 -- Enqueue flights to all unmarked adjacent cities
 for Index in 1..List.Length loop
 Next_City := List.Items(Index); -- Name of adjacent city
 if not Air_Routes.Marked (Graph, Next_City) then
 -- Set up the flight to Next_City
 Flight.Route := (From => Current_City, To => Next_City);
 -- The distance from Start_City to Next_City is the sum of the
 -- distance from Start_City to Current_City and the distance
 -- between Current_City and Next_City.
 Flight.Total := Distance + Air_Routes.Weight_Of (Graph, Flight.Route);
 P_Queue.Enqueue (Queue => Queue, Item => Flight);
 end if;
 end loop;
 end if; -- If not marked
end loop City_Loop;
end Shortest_Paths;

```

The output from this procedure is a table of city pairs (edges) showing the total distance from `Start_City` to each of the other vertices in the graph, as well as the last vertex visited before the destination. If `Graph` contains the information shown in Figure 13-13 (page 618), the procedure call

```
Shortest_Paths (Graph => Graph, Start_City => "Washington");
```

prints out the following table:

Shortest Path from Washington

| Last City  | Destination | Distance |
|------------|-------------|----------|
| Washington | Washington  | 0        |
| Washington | Atlanta     | 600      |
| Washington | Dallas      | 1300     |
| Atlanta    | Houston     | 1400     |
| Dallas     | Austin      | 1500     |
| Dallas     | Denver      | 2080     |
| Dallas     | Chicago     | 2200     |

The shortest-path distance from Washington to each destination is shown in the two columns to the right. For example, our flights from Washington to Chicago total 2200 miles. The left-hand column shows which city immediately preceded the destination in the traversal. Let's figure out the shortest path from Washington to Chicago. We see from the left-hand column that the next-to-last vertex in the path is Dallas. Now we look up Dallas in the Destination (middle) column: the vertex before Dallas is Washington. The whole path is Washington-Dallas-Chicago.

### The Implementation Level

#### Adjacency Matrix

For a graph with  $N$  nodes, an  $N \times N$  table that shows the existence (or weights) of all edges in the graph.

### An Array-Based Implementation

A simple way to represent the edges in a graph is by using an **adjacency matrix**, a two-dimensional array of edge values. We use the following declarations for the private part of Specification 13-3 to implement a weighted graph:

```
private

type Vertex_Rec is
 record
 Info : Vertex_Type; -- Information declared by user
 Marked : Boolean := False; -- Has the Vertex been visited?
 end record;

type Edge_Rec is
 record
 Weight : Weight_Type; -- Weight assigned to the edge
 Defined : Boolean := False; -- Is this edge in the graph?
 end record;
```

```

type Vertex_Array is array (Positive range <>) of Vertex_Rec;
type Matrix_Type is array (Positive range <>,
 Positive range <>) of Edge_Rec;

type Graph_Type (Max_Vertices : Positive) is
 record
 Num_Vertices : Natural := 0;
 Vertices : Vertex_Array (1..Max_Vertices);
 Edges : Matrix_Type (1..Max_Vertices, 1..Max_Vertices);
 end record;

end Graph;

```

A graph consists of count of vertices, a one-dimensional array of `Vertex_Rec`, and a two-dimensional array of `Edge_Rec`. Figure 13-18 depicts the implementation of the graph of Air Busters flights between seven cities. For simplicity, only the user information (elements and weights) is shown; the Boolean fields are omitted. (A drawing of this graph is found in Figure 13-13, page 618.) The names of the cities are contained in `Graph.Vertices`. The city names in Figure 13-18 are in alphabetical order, but there is no requirement that the elements in this array be ordered.

The weight of each edge in the Edge adjacency matrix represents the air distance between two cities that are connected by a flight. For example, the value in `Graph.Edges (2, 4)` tells us that there is a direct flight from Austin to Dallas, and that the air distance is 200 miles. The value in `Graph.Edges (4, 2)` tells us that there is also a direct flight from Dallas to Austin.

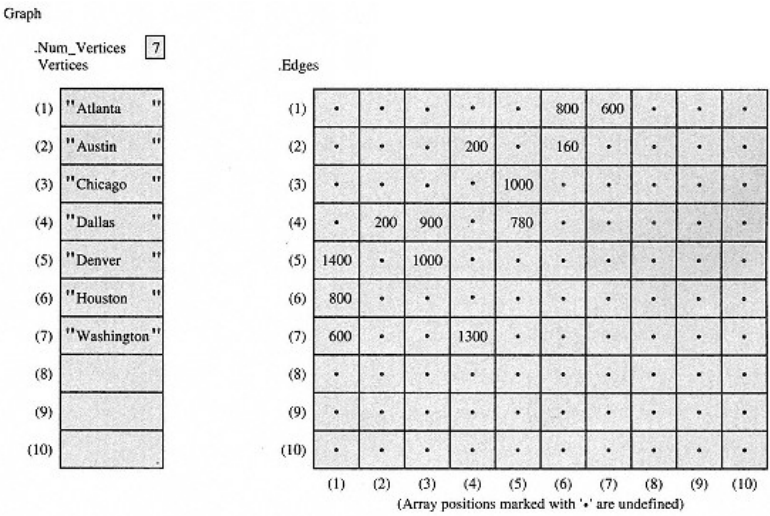


Figure 13-18  
Adjacency Matrix Representation of a Graph of Flight Connections Between Cities



As you can guess from its name, the Boolean field `Marked` in `Vertex_Rec` is used to indicate whether or not the user has marked the vertex. The adjacency matrix, `Graph.Edges`, contains an element for every *possible* edge in the graph. Usually, some of these edges are not defined. The Boolean field `Defined` in `Edge_Rec` indicates whether or not the edge represented by a particular row and column in the matrix is defined in the graph. In Figure 13-18, undefined edges are represented by a dot. We can see from this figure that there is no flight (edge) from Atlanta to Dallas.

Now let's look at how the graph operations specified earlier would be implemented using these declarations. The `Clear` procedure sets `Graph` to its empty state. By setting the `Num_Vertices` field to zero, we indicate that there are no vertices in `Graph.Vertices` and that the part of `Graph.Edges` that is in use is a  $0 \times 0$  subarray; that is, there are no edges.

The `Add_Vertex` procedure puts `Vertex` in the next free space in the `Vertices` array. Because the new vertex has no edges defined yet, we also initialize the `Defined` field of all the array elements in the appropriate row and column in the `Edges` matrix to `False`.

To add an edge to the graph, we must first locate the `From` vertex and `To` vertex given in the record parameter `Edge` that define the edge we want to add. To index the correct matrix slot, we need the index in the `Vertices` array that corresponds to each vertex. Once we know the indexes, it's a simple matter to set the weight of the edge in the matrix and set the `Defined` field to `True`. Here is the algorithm:

#### Add Edge

```
From_Index := Index of Edge.From in Graph.Vertices
To_Index := Index of Edge.To in Graph.Vertices
Graph.Edges (From_Index, To_Index).Weight := Weight
Graph.Edges (From_Index, To_Index).Defined := True
```

We write a search function, `Index_Of`, that is given the key of a vertex and returns its location (`Index`) in `Graph.Vertices`. Here is the declaration of this local function:

```
function Index_Of (Graph : in Graph_Type;
 Key : in Key_Type) return Positive;
-- Returns the index of the vertex in Vertices with the given Key

-- Preconditions: None

-- Postconditions: If there is an element in Graph with the given Key
-- The index of it in Graph.Vertices is returned
-- Else
-- Graph.Num_Vertices + 1 is returned
```

Procedure `Retrieve` returns the information associated with the vertex with the given `Key`. We again use function `Index_Of` to find the desired vertex in `Graph.Vertices`. Once it is found, its `Info` field is returned.

The `Weight_Of` function returns the weight of an `Edge`. To implement this function, we first use our search function `Index_Of` to find the indexes of the two vertices (given by the `To` and the `From` fields of the edge record); then we return the value from the appropriate slot in the `Edges` matrix.

The three graph operations `Clear_All_Marks`, `Mark_Vertex`, and `Marked` all modify or examine the `Marked` field in the vertices in the array of vertices, `Graph.Vertices`. `Clear_All_Marks` traverses the one-dimensional array, setting every `Marked` field to `False`. We use our search function, `Index_Of`, to locate the particular vertex to check or modify.

The last graph operation that we specified is `Get_Adjacent_Vertices`. This procedure inputs a graph and the `Key` of a vertex and outputs a list of vertex keys that are adjacent from the designated vertex. That is, it returns a list of all the vertices that you can get to from this vertex. Using an adjacency matrix to represent the edges, it's a simple matter to determine which nodes are adjacent to `Vertex`. We merely loop through the appropriate row in `Graph.Edges`; whenever a `Defined` value is found, we add another vertex key to the list. The complete package body for the adjacency matrix implementation of `Graph` is given in Body 13-6.

#### Body 13-6—

#### Weighted Graph Implements Specification 13-3

package body `Graph` is

```

function Index_Of (Graph : in Graph_Type;
 Key : in Key_Type) return Positive is
-- Returns the index of the vertex in Vertices with the given Key

-- Preconditions: None

-- Postconditions: If there is an element in Graph with the given Key
-- The index of it in Graph.Vertices is returned,
-- Else
-- Graph.Num_Vertices - 1 is returned.
 Index : Positive;
begin
 Index := 1;
 loop
 exit when Index > Graph.Num_Vertices or else
 Equal (Key, Key_Of (Graph.Vertices(Index).Info));
 Index := Index + 1;
 end loop;
 return Index;
end Index_Of;

procedure Clear (Graph : in out Graph_Type) is
begin
 Graph.Num_Vertices := 0;
end Clear;

```

```

procedure Add_Vertex (Graph : in out Graph_Type;
 Vertex : in Vertex_Type) is
 Index : Positive; -- Location for new Vertex
begin
 Index := Index_Of (Graph, Key_Of (Vertex)); -- Find if its key is already
 if Index <= Graph.Num_Vertices then -- in the Graph
 raise VERTEX_ERROR;
 elsif Index > Graph.Max_Vertices then -- Any more room?
 raise OVERFLOW;
 else
 Graph.Vertices (Index).Info := Vertex; -- Add the new vertex to the
 Graph.Vertices (Index).Marked := False; -- end of the array
 Graph.Num_Vertices := Index;
 -- Initialize the adjacency matrix to indicate no edges
 for Row in 1..Graph.Num_Vertices loop -- The row
 Graph.Edges (Row, Index).Defined := False;
 end loop;
 for Col in 1..Graph.Num_Vertices loop -- The column
 Graph.Edges (Index, Col).Defined := False;
 end loop;
 end if;
end Add_Vertex;

```

```

procedure Add_Edge (Graph : in out Graph_Type;
 Edge : in Edge_Type;
 Weight : in Weight_Type) is
 From_Index : Positive;
 To_Index : Positive;
begin
 From_Index := Index_Of (Graph, Edge.From); -- Location of From vertex
 To_Index := Index_Of (Graph, Edge.To); -- Location of To vertex
 -- Do both vertices exist?
 if From_Index > Graph.Num_Vertices or To_Index > Graph.Num_Vertices then
 raise VERTEX_ERROR;
 else
 -- Update the adjacency matrix with the new edge
 Graph.Edges (From_Index, To_Index).Weight := Weight;
 Graph.Edges (From_Index, To_Index).Defined := True;
 end if;
end Add_Edge;

```

```

procedure Retrieve (Graph : in Graph_Type;
 Key : in Key_Type;
 Vertex : out Vertex_Type) is
 Index : Positive;
begin
 Index := Index_Of (Graph, Key); -- Find the vertex's location
 if Index > Graph.Num_Vertices then
 raise VERTEX_ERROR; -- Vertex does not exist
 else
 Vertex := Graph.Vertices (Index).Info; -- Return information
 end if;
end Retrieve;

```

```

 end if;
end Retrieve;

```

```

function Weight_Of (Graph : in Graph_Type;
 Edge : in Edge_Type) return Weight_Type is
 From_Index : Positive;
 To_Index : Positive;
begin
 From_Index := Index_Of (Graph, Edge.From); -- Find the location of
 To_Index := Index_Of (Graph, Edge.To); -- the two vertices in Edge
 -- Check to make sure that both vertices exist in Graph
 if From_Index > Graph.Num_Vertices or To_Index > Graph.Num_Vertices then
 raise VERTEX_ERROR;
 elsif Graph.Edges(From_Index, To_Index).Defined then -- Is the edge defined?
 return Graph.Edges(From_Index, To_Index).Weight;
 else
 raise EDGE_ERROR;
 end if;
end Weight_Of;

```

```

procedure Get_Adjacent_Vertices (Graph : in Graph_Type;
 Key : in Key_Type;
 Adj_Keys : out Key_List) is
 From_Index : Positive;
 List : Key_List (Adj_Keys.Max_Vertices); -- Local copy of out parameter
begin
 From_Index := Index_Of (Graph, Key); -- Locate the vertex with the given Key
 if From_Index > Graph.Num_Vertices then
 raise VERTEX_ERROR; -- Vertex does not exist
 else
 List.Length := 0; -- Initialize list of adjacent vertex keys
 -- Go through the From_Index row of the matrix
 for To_Index in 1..Graph.Num_Vertices loop
 -- Is there an edge from From_Index to To_Index?
 if Graph.Edges(From_Index, To_Index).Defined then
 -- Add the key of the To end of the edge to the list
 List.Length := List.Length + 1;
 List.Items(List.Length) := Key_Of (Graph.Vertices(To_Index).Info);
 end if;
 end loop;
 Adj_Keys := List; -- Copy result to out parameter
 end if;
end Get_Adjacent_Vertices;

```

```

procedure Clear_All_Marks (Graph : in out Graph_Type) is
begin
 for Index in 1..Graph.Num_Vertices loop
 Graph.Vertices(Index).Marked := False;
 end loop;
end Clear_All_Marks;

```

```

procedure Mark_Vertex (Graph : in out Graph_Type;
 Key : in Key_Type) is
 Index : Positive;
begin
 Index := Index_Of (Graph, Key); -- Find location of vertex with Key
 if Index > Graph.Num_Vertices then
 raise VERTEX_ERROR;
 else
 Graph.Vertices(Index).Marked := True;
 end if;
end Mark_Vertex;

function Marked (Graph : in Graph_Type;
 Key : in Key_Type) return Boolean is
 Index : Positive;
begin
 Index := Index_Of (Graph, Key); -- Find location of vertex with Key
 if Index > Graph.Num_Vertices then
 raise VERTEX_ERROR;
 else
 return Graph.Vertices(Index).Marked;
 end if;
end Marked;

end Graph;

```

---

**Adjacency List**

A linked list that identifies all the vertices to which a particular vertex is connected. Each vertex has its own adjacency list.

---

### Linked Implementations

The advantages to representing the edges in a graph with an adjacency matrix are its speed and simplicity. Given the indexes of two vertices, determining the existence (or the weight) of an edge between them is an  $O(1)$  operation. The problem with adjacency matrices is that their use of *space* is  $O(N^2)$ , where  $N$  is the *maximum* number of vertices in the graph. If the maximum number of vertices is large, adjacency matrices may waste a lot of space. In the past, we have tried to save space by allocating it as we need it at run time, using linked structures. We can use a similar approach to implementing graphs. **Adjacency lists** are linked lists, one per vertex, that identify the vertices to which each vertex is connected. There are several ways to implement adjacency lists. Figure 13-19 shows two different adjacency list representations of the graph in Figure 13-13.

In Figure 13-19(a), the list of vertices is stored in an array. Each component of this array contains a pointer to a linked list of edge nodes. Each node in these linked lists contains an index number, a weight, and a pointer to the next node in the adjacency list. Let's look at the adjacency list for Denver. The first node in the list indicates that there is a 1400 mile flight from Denver to Atlanta (the vertex whose index is 1) and a 1000 mile flight from Denver to Chicago (the vertex whose index is 3).

No arrays are used in the implementation illustrated in Figure 13-19(b) on page 640. The list of vertices is implemented as a linked list. Now each node in the adjacency lists contains a pointer to the vertex information rather than the index of the vertex. Because there are so many of these pointers in Figure 13-19(b), we have used text to describe the vertex that each pointer designates rather than drawing them as arrows.

The advantage of adjacency lists over the adjacency matrix is the savings in memory. The price we pay for this memory is the extra time it takes to access a particular edge value. With the matrix, the two array indices allow us to access edge information directly. To access this information in an adjacency list, we must search the list. The declarations for a graph based on adjacency lists and the implementation of the graph operations are left as exercises.

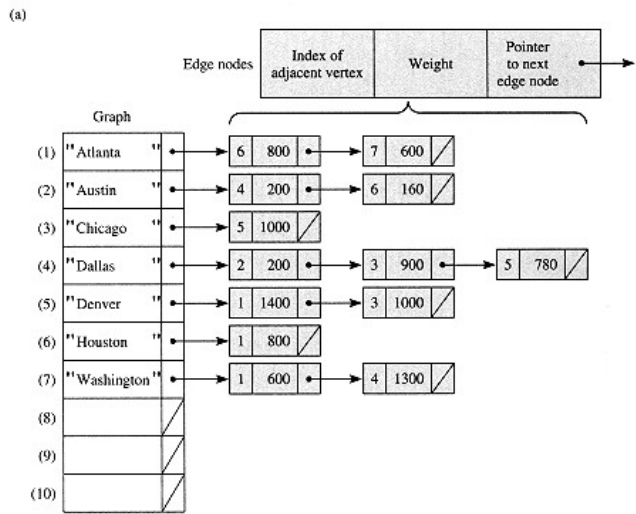


Figure 13-19  
Adjacency List Representations of Graphs

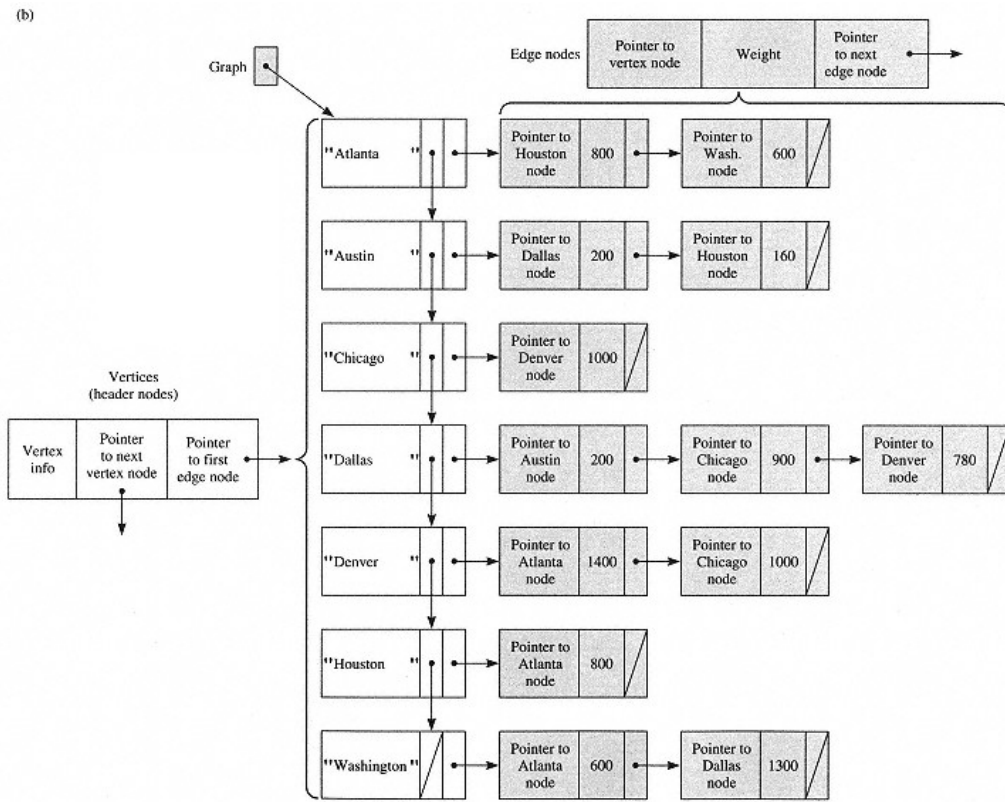


Figure 13-19  
(continued)

## Summary

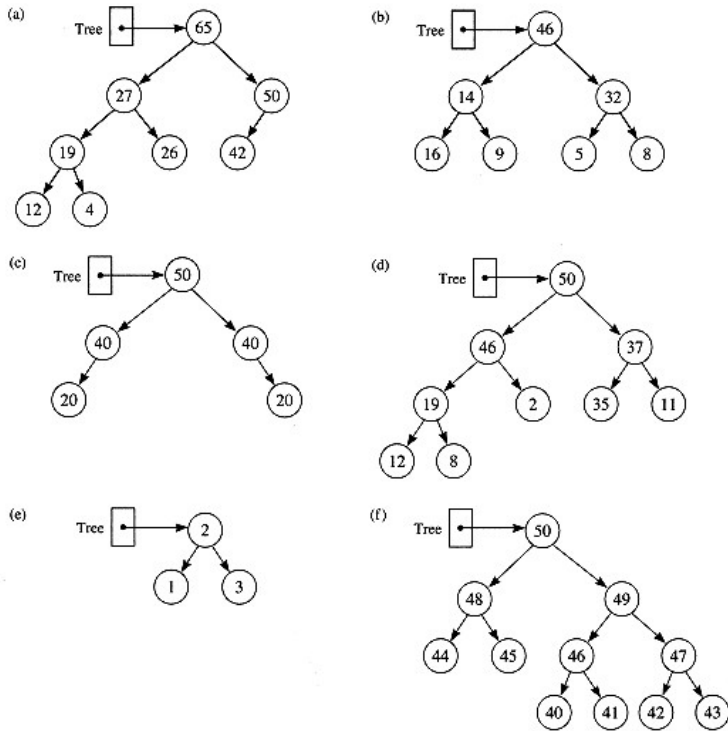
In this chapter we discussed several branching structures: trees, heaps, and graphs. Branching structures are very versatile and are a good way to model many real-world objects and situations. Because there are many different types of applications for these data structures, there are all kinds of variations and generalizations of trees and graphs. We introduced these topics here in order to show the wide variety of applications for which programmers must create appropriate data structures. They are generally covered in detail in more advanced computer science courses.

## Exercises

1. (a) Which of the following trees are complete?

(b) Which of the following trees are full?

(c) Which of the following trees are heaps?



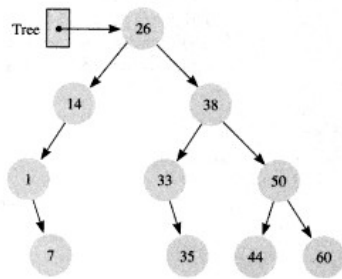


2. The elements in a binary tree are to be stored in an array, as described in the chapter. Each element is a `Natural` value.

(a) What value can you use as the dummy value if the binary tree is not complete?

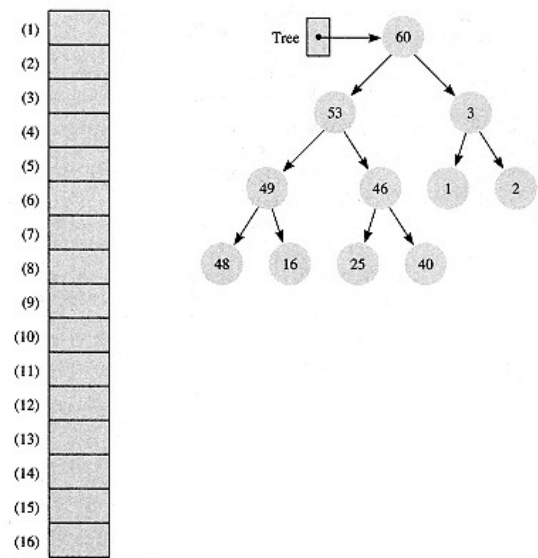
(b) Show the contents of the array, given the tree illustrated below.

|      |  |
|------|--|
| (1)  |  |
| (2)  |  |
| (3)  |  |
| (4)  |  |
| (5)  |  |
| (6)  |  |
| (7)  |  |
| (8)  |  |
| (9)  |  |
| (10) |  |
| (11) |  |
| (12) |  |
| (13) |  |
| (14) |  |
| (15) |  |
| (16) |  |



3. The elements in a complete binary tree are to be stored in an array, as described in the chapter. Each element is a `Natural` value.

- (a) Show the contents of the array, given the tree illustrated below.
- (b) Is this tree a heap?



4. (a) Given the array pictured below, draw the binary tree that can be created from its elements. (The elements are arranged in the array as discussed in the chapter.)

- (b) Is this tree a heap? If not, what procedure call(s) can be used to make the tree into a heap?

Num\_Elements = 9

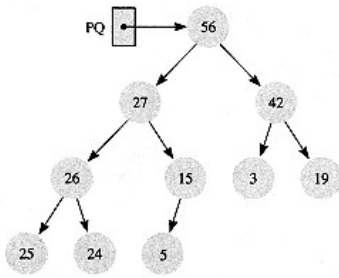
|      |    |
|------|----|
| (1)  | 1  |
| (2)  | 55 |
| (3)  | 59 |
| (4)  | 44 |
| (5)  | 33 |
| (6)  | 58 |
| (7)  | 57 |
| (8)  | 22 |
| (9)  | 11 |
| (10) | 99 |

5. Draw a tree that satisfies both the binary search property and the order property of heaps.
6. A binary tree is stored in an array called `Tree`, which is indexed from 1 to 100, as described in the chapter. The tree contains 85 elements. Mark each of the following statements as True or False, and correct any false statements.
- (a) `Tree (43)` is a leaf node.
  - (b) `Tree (42)` has only one child.
  - (c) The right child of `Tree (13)` is `Tree (26)`.
  - (d) The subtree rooted at `Tree (8)` is a full binary tree with four levels.
  - (e) The tree has seven levels that are full, and one additional level that contains some elements.
7. A *minimum heap* has the following order property: The value of each element is less than or equal to the value of each of its children. How can we instantiate a minimum heap package from the maximum heap package we developed in this chapter?
8. (a) Write a nonrecursive version of `Reheap_Down`.
- (b) Write a nonrecursive version of `Reheap_Up`.
  - (c) Describe the nonrecursive versions of these operations in terms of Big-O.
9. A priority queue containing uppercase alphabetic characters is implemented as a heap stored in an array. The precondition states that this priority queue cannot contain duplicate elements. There are 10 elements currently in the priority queue. What values might be stored in array positions 8-10 so that the properties of a heap are satisfied?

| Heap |   |
|------|---|
| (1)  | Z |
| (2)  | F |
| (3)  | J |
| (4)  | E |
| (5)  | B |
| (6)  | G |
| (7)  | H |
| (8)  | ? |
| (9)  | ? |
| (10) | ? |

10. A student record contains a name (a string of no more than 20 characters), a GPA (a float value between 0.0 and 4.0), and the number of days of class missed.
- (a) Write the necessary Ada code to instantiate a priority queue of student records. Students with higher GPAs are to have higher priority.
  - (b) Write the necessary Ada code to instantiate a priority queue of student records. Students with fewer missed class days are to have higher priority.

11. A priority queue is implemented as a heap.



(a) Show how the heap above would look after this series of operations:

Priority\_Queue.Enqueue (PQ, 28);

Priority\_Queue.Enqueue (PQ, 2);

Priority\_Queue.Enqueue (PQ, 40);

Priority\_Queue.Dequeue (PQ, X);

Priority\_Queue.Dequeue (PQ, Y);

Priority\_Queue.Dequeue (PQ, Z)

(b) What would the values of X, Y, and Z be after the series of operations in part (a)?

12. A priority queue is implemented as a linked list, ordered from largest to smallest element.

(a) Write the declarations in the private part of the priority queue package declaration needed for this implementation.

(b) Write the Enqueue operation (from the chapter specification) using this implementation.

(c) Write the Dequeue operation (from the chapter specification) using this implementation.

(d) Compare the Enqueue and Dequeue operations to those for the heap implementation in terms of Big-O.

13. A priority queue is implemented as a binary search tree.

(a) Write the declarations in the private part of the priority queue package declaration needed for this implementation.

(b) Write the Enqueue operation (from the chapter specification) using this implementation.

(c) Write the Dequeue operation (from the chapter specification) using this implementation.

(d) Compare the Enqueue and Dequeue operations to those for the heap implementation in terms of Big-O. Under what conditions would this implementation be better or worse than the heap implementation?

14. A priority queue is implemented as a sequential array-based list. The highest priority item is in the first array position, the second highest priority is in the second array positions, and so on.

(a) Write the declarations in the private part of the priority queue package declaration needed for this implementation.

(b) Write the Enqueue operation (from the chapter specification) using this implementation.

(c) Write the Dequeue operation (from the chapter specification) using this implementation.

(d) Compare the Enqueue and Dequeue operations to those for the heap implementation in terms of Big-O. Under what conditions would this implementation be better or worse than the heap implementation?

15. A stack is implemented using a priority queue. Each element is time-stamped as it is put into the stack. (The time stamp is a number between zero and `Natural'Last`. Each time an element is pushed onto the stack, it's assigned the next larger number.)

- What is the highest-priority element?
- Write the necessary Ada code to instantiate a suitable priority queue from the package developed in this chapter.
- Write the Push and Pop algorithms using the specifications in Chapter 6.
- Compare these Push and Pop operations to the ones implemented in Chapter 6 in terms of Big-O.
- Is this priority-queue-based solution a reasonable way to implement a stack?

16. A FIFO queue is implemented using a priority queue. Each element is time stamped as it is put into the queue. (The time stamp is a number between 0 and `Natural'Last`. Each time an element is Enqueued, it is assigned the next larger number.)

- What is the highest-priority element?
- Write the necessary Ada code to instantiate a suitable priority queue from the package developed in this chapter.
- Write the Enqueue and Dequeue algorithms using the specifications in Chapter 7.
- Compare these Enqueue and Dequeue algorithms to the ones implemented in Chapter 7 in terms of Big-O.

17. A priority queue of strings is implemented using a heap. The heap contains the following elements.

Num\_Elements = 10

|      |                 |
|------|-----------------|
| (1)  | "introspective" |
| (2)  | "intelligent"   |
| (3)  | "intellectual"  |
| (4)  | "intimate"      |
| (5)  | "intensive"     |
| (6)  | "interesting"   |
| (7)  | "internal"      |
| (8)  | "into"          |
| (9)  | "in"            |
| (10) | "intro"         |

- What feature of these strings is used to determine their priority in the priority queue?
- Show how this priority queue is affected adding the string "interviewing".

18. Assuming that the strings in the previous question are implemented with the bounded string package given in Chapter 5, write the necessary Ada code to instantiate a priority queue that satisfies the requirements of the previous question.

19. Figure 13-16 shows the contents of the queue during part of the breadth-first search of Austin to Washington. Draw the queue for the remaining iterations of the search loop.

Use the following description of an *undirected graph* in Exercises 20-23.

Employee\_Graph = (V, E)

$V(\text{Employee\_Graph}) = \{\text{Susan, Darlene, Mike, Fred, John, Sander, Lance, Jean, Brent, Fran}\}$

$E(\text{Employee\_Graph}) = \{(\text{Susan, Darlene}), (\text{Fred, Brent}), (\text{Sander, Susan}), (\text{Lance, Fran}), (\text{Sander, Fran}), (\text{Fran, John}), (\text{Lance, Jean}), (\text{Jean, Susan}), (\text{Mike, Darlene}), (\text{Brent, Lance}), (\text{Susan, John})\}$

20. Draw a picture Employee\_Graph.

21. Draw Employee\_Graph, implemented as an adjacency matrix. Store the vertex values in alphabetical order.

22. Using the adjacency matrix for Employee\_Graph from Exercise 21, describe the path from Susan to Lance

- (a) using a breadth-first strategy.
- (b) using a depth-first strategy.

23. Which one of the following phrases best describes the relationship represented by the edges between the vertices in Employee\_Graph?

- (a) "works for"
- (b) "is the supervisor of"
- (c) "is senior to"
- (d) "works with"

Use the following specification of a *directed graph* in Exercises 24-27:

Zoo\_Graph = (V, E)

$V(\text{Zoo\_Graph}) = \{\text{dog, cat, animal, vertebrate, oyster, shellfish, invertebrate, crab, poodle, monkey, banana, dalmation, dachshund}\}$

$E(\text{Zoo\_Graph}) = \{(\text{vertebrate, animal}), (\text{invertebrate, animal}), (\text{dog, vertebrate}), (\text{cat, vertebrate}), (\text{monkey, vertebrate}), (\text{shellfish, invertebrate}), (\text{crab, shellfish}), (\text{oyster, shellfish}), (\text{poodle, dog}), (\text{dalmation, dog}), (\text{dachshund, dog})\}$

24. Draw a picture of Zoo\_Graph.

25. Draw the adjacency matrix for Zoo\_Graph. Store the vertices in alphabetical order.

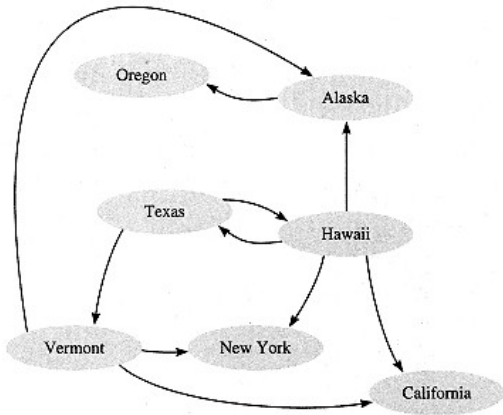
26. To tell if one element in Zoo\_Graph has relation X to another element, you look for a path between them. Show whether the following statements are true, using the picture or adjacency matrix:

- (a) dalmation X dog
- (b) dalmation X vertebrate
- (c) dalmation X poodle
- (d) banana X invertebrate
- (e) oyster X invertebrate
- (f) monkey X invertebrate

27. Which of the following phrases best describes relation X in the previous question?

- (a) "has a"
- (b) "is a kind of"
- (c) "is a generalization of"
- (d) "eats"

Use the following graph for Exercises 28-30.



28. Describe the graph pictured above, using the formal graph notation.

$V(\text{State\_Graph}) =$

$E(\text{State\_Graph}) =$

29. (a) Is there a path from Oregon to any other state in the graph?

(b) Is there a path from Hawaii to every other state in the graph?

(c) From which state(s) in the graph is there a path to Hawaii?

30. Show the adjacency matrix that would describe the edges in this graph. Store the vertices in alphabetical order.

31. We are going to extend the Graph package in this chapter to include the following Boolean functions: `Full` (determines if a graph's set of vertices is "full"), `Empty` (determines if a graph has no vertices), `Edge_Exists` (determines if two vertices are connected by an edge).

(a) Write the declarations of these three functions. Include adequate comments.

(b) Using the adjacency matrix implementation developed in the chapter, implement the body for these three functions.

32. We are going to extend the Graph package in this chapter to include a `Delete_Edge` operation, which deletes the given edge.

(a) Write the declaration of this operation. Include adequate comments.

(b) Using the adjacency matrix implementation developed in the chapter and the declaration from part (a), implement the body of the procedure.

33. We are going to extend the Graph package in this chapter to include a `Delete_Vertex` operation, which deletes a vertex from the graph.

(a) Write the declaration of this operation. Include adequate comments.

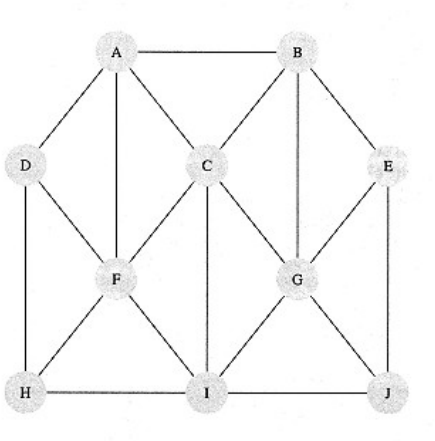
(b) Deleting a vertex is more complicated than deleting an edge from the graph. Discuss why.

34. Implement the `Delete_Vertex` operation described in the previous question.

35. The `Depth_First_Search` operation can be implemented without a stack by using recursion.

- Name the base case(s). Name the general case(s).
- Write the algorithm for a recursive depth-first search.
- Code the recursive procedure using the `Air_Routes` graph package instantiated in this chapter.

Use the following graph for Exercises 36-38. The vertices are single-character. The graph is undirected.



The graph is to be implemented as an adjacency matrix indexed by the characters 'A' through 'Z', using the following declarations:

```
subtype Alpha_Type is Character range 'A'..'Z';

type Graph_Type is array (Alpha_Type, Alpha_Type) of Boolean;

Alpha_Graph : Graph_Type;
```

36. Show the contents of `Alpha_Graph`, given the graph pictured above. (You only need to show the subarray that is in use.)

Note: The edge (D,F) is represented by `Alpha_Graph('D', 'F')` and `Alpha_Graph('F', 'D')`.

37. Reimplement the graph operations specified in the chapter to use the declarations above.

38. Implement a graph operation to delete a vertex from the graph. The edges to all adjacent vertices must also be deleted.

### Programming Problems

1. Write a program to schedule the jobs submitted to a college's computer center printer. Your program should read all of the print requests from a file. Each print request consists of title



(Administrator, Professor, Student) and the size (in bytes) of the print job. Enqueue each print request into a priority queue where student jobs have high priority, professor jobs have medium priority, and administrative jobs have low priority. Within each of these three groups, smaller jobs go before larger jobs. Thus a student job to print 12,434 bytes has a higher priority over a student job to print 52,198 bytes. After enqueueing all of the print requests in the file, dequeue them all and display the job characteristics of each (Title and Size).

2. Your assignment is to compare two implementations of a priority queue whose highest priority element is the one with the smallest key value. The elements have the following declarations:

```
subtype Name_String is String (1..10);
```

```
type Element_Type is
 record
 Key : Positive;
 Name : Name_String;
 end record
```

### ***The Priority Queue Implementations***

Use an instantiation of the heap-based priority queue ADT developed in this chapter for the first implementation. For the second implementation, use a linear linked list whose elements are ordered by key value.

### ***Testing the Implementations***

After the operations for each implementation are written, test them with a batch test driver that performs the following operations:

```
Enqueue 43, Robert
Enqueue 5, Miriam
Enqueue 8, Joshua
Enqueue 36, Susan
Enqueue 11, David
Enqueue 1, Leah
Dequeue
Dequeue
Enqueue 2, Suzanne
Enqueue 5, John
Enqueue 10, Nell
Dequeue all remaining elements
```

After each Dequeue operation, print the Key and Name of the element returned. Print all output to both the screen and a file called `PQUEUE.OUT`.

### ***Comparing the Implementations***

To compare the operations, you must modify the Enqueue and Dequeue operations to count and return how many elements are accessed (compared or swapped, in the case of reheaping) during its execution. Run the test cases described in the previous section, recording the number of elements accessed by each operation.

### ***Turn In***

1. source listing of each priority queue implementation
2. source listing of the test driver
3. report comparing the number of elements accessed in executing each operation

3. Reimplement the Graph ADT using an adjacency list representation. Figure 13-19 suggests two ways that the graph could be implemented. You should implement the originally specified Graph operations and the Mark operations.

### ***Testing the Graph Operations***

Test the graph operations with a test driver that builds the graph depicted in Figure 13-13, and performs the following depth-first and breadth-first searches:

Houston to Denver

Washington to Chicago

Austin to Washington

All output should be directed to both the screen and file `GRAPH.OUT`.

### ***Turn In***

1. source listing of the graph package declaration
2. source listing of the graph package body
3. source listing of test driver
4. hard copy of file `GRAPH.OUT`, containing the output from the test driver

**14—****Sorting Algorithms****GOALS**

After reading this chapter, you should be able to

- design and implement the following sorting algorithms:

straight selection sort

bubble sort

insertion sort

merge sort

quick sort

heap sort

radix sort

- compare the efficiency of the sorting algorithms in terms of Big-O and space requirements
- discuss other efficiency considerations: sorting small numbers of elements, programmer time, sorting arrays of large data elements
- sort on several keys

At many points in this book, we have gone to great trouble to keep lists of elements in sorted order: student records sorted by ID number, integers sorted from smallest to largest, words sorted alphabetically. The goal of keeping sorted lists, of course, is to facilitate searching. Given an appropriate data structure, a particular list element can be found faster if the list is sorted.

In Chapters 14 and 15 we examine strategies for sorting and searching: two tasks that are fundamental to a variety of computing problems. Because more computer time is spent sorting and searching than for any other task, it's important that we examine these topics in some detail.

### A Generic Sort Declaration

Putting an unordered list of data items into order—**sorting**—is a very common and useful operation. Whole books have been written about various sorting algorithms, as well as algorithms for searching an ordered list to find a particular element. The goal is to come up with better, *more efficient*, sorts. Because sorting a large number of elements can be extremely time consuming, a good sorting algorithm is very desirable.

How do we describe efficiency? We pick an operation central to most sorting algorithms: the operation that compares two values to see which is smaller. In our study of sorting algorithms, we relate the number of comparisons to the number of elements in the list ( $N$ ) as a rough measure of the efficiency of each algorithm. The number of swaps made is another measure of sorting efficiency. Exercise 36 asks you to analyze the sorting algorithms developed in this chapter in terms of moving data.

Another efficiency consideration is the amount of memory space required. In general, memory space is not a very important factor in choosing a sorting algorithm. We look at only two sorts in which space would be a serious consideration. The usual time versus space trade-off applies to sorts—more space often means less time, and vice versa.

Because processing time is the factor that applies most often to sorting algorithms, we have considered it in detail here. Of course, as in any application, the programmer must determine goals and requirements before selecting an algorithm and starting to code.

We review the straight selection sort, the bubble sort, and the insertion sort. You probably wrote at least one of these three simple sorts in your first course. Then we examine three more complex sorting algorithms: merge sort, quick sort, and heap sort. We implement each of these six sorting algorithms as a generic procedure using the declarations given in Specification 14-1. Finally we look at the radix sort, a completely different approach to sorting.

### Specification 14-1— Sort

generic

```
type Element_Type is private;
type Array_Type is array (Positive range <>) of Element_Type;
with function "<" (Left : in Element_Type; Right : in Element_Type) return Boolean;
```

```
procedure Some_Sort (Info : in out Array_Type);
```

```
-- Sorts the elements in List from smallest key to largest key
-- as determined by function "<"

-- Preconditions: Only one of our sorts (Heap Sort) has preconditions

-- Postconditions: The elements in Info are in order by function "<"
```

This generic procedure sorts an array of elements. What we usually need to sort are the elements in an array-based list such as the one implemented by these declarations:

```
type List_Array is array (Positive range <>) of Element_Type;

type List_Type (Max_Size : Positive) is
 record
 Length : Natural := 0;
 Info : List_Array (1..Max_Size);
 end record;

List : List_Type;
```

Because the elements in `List` are stored in an array, we sort `List` by passing the slice of the array containing the actual elements. Here is an example.

```
-- Sort the elements in List
An_Instantiated_Sort (Info => List.Info (1..List.Length));
```

## **$O(N^2)$ Sorts**

### ***Straight Selection Sort***

If you were handed a list of names and asked to put them in alphabetical order, you might use this general approach:

1. Find the name that comes first in the alphabet, and write it on a second sheet of paper.
2. Cross the name out on the original list.
3. Continue this cycle until all the names on the original list have been crossed out and written onto the second list, at which point the second list is sorted.

This algorithm is simple to translate into a computer program, but it has one drawback: It requires space in memory to store two complete lists. Although we have not talked much about memory space considerations, this duplication is clearly wasteful. A slight adjustment to this manual approach does away with the need to duplicate space, however. As you cross a name off the original list, a free space opens up. Instead of writing the minimum value on a second list, you can exchange it with the value currently in the position where the crossed-off element would go. Let's look at an example: sorting the five-element array shown in Figure 14-1(a).

Because of this algorithm's simplicity, it's usually the first sorting method that students learn. Therefore, we go straight to the algorithm.

1. The smallest value in `Info` is in location (4). Therefore we swap the contents of `Info(1)` and `Info(4)`. Now the smallest value is in its correct position in the array [Figure 14-1(b)].
2. The smallest value in the unsorted part of the array (`Info(2) . . . Info(5)`) is in location (3), so we swap `Info(2)` and `Info(3)`. Now the first two positions in the array are sorted [Figure 14-1(c)].
3. The smallest value in the unsorted part of the array (`Info(3) . . . Info(5)`) is in `Info(3)`, its correct position, so we do nothing [Figure 14-1(d)].
4. The smallest value in the unsorted part of the array (`Info(4) . . . Info(5)`) is in position (5), so we swap `Info(4)` and `Info(5)`. The only element left is `Info(5)`, which must be the largest value in the array. We now have a sorted array, as shown in Figure 14-1(e).

| Info |     | Info |     | Info |     | Info |     | Info |     |
|------|-----|------|-----|------|-----|------|-----|------|-----|
| (1)  | 126 | (1)  | 1   | (1)  | 1   | (1)  | 1   | (1)  | 1   |
| (2)  | 43  | (2)  | 43  | (2)  | 26  | (2)  | 26  | (2)  | 26  |
| (3)  | 26  | (3)  | 26  | (3)  | 43  | (3)  | 43  | (3)  | 43  |
| (4)  | 1   | (4)  | 126 | (4)  | 126 | (4)  | 126 | (4)  | 113 |
| (5)  | 113 | (5)  | 113 | (5)  | 113 | (5)  | 113 | (5)  | 126 |
| (a)  |     | (b)  |     | (c)  |     | (d)  |     | (e)  |     |

Figure 14-1  
Example of Straight Selection Sort (The sorted elements are shaded.)

Let's see how this approach can be described in terms of a programming algorithm:

### Selection Sort

```

Current := Info'First
loop
 exit when there is only one element left in the unsorted part of the array
 Find the index of the smallest unsorted element
 Swap the Current element and the smallest unsorted element
 Shrink the unsorted part of the array by incrementing Current
end loop

```

We use a variable, `Current`, to mark the beginning of the unsorted part of the array. We start out by setting `Current` to the index of the first array position. That means that the unsorted part of the array is `Info(Current) . . Info'Last`.

The main sort processing is a loop. In each iteration of the loop body, the smallest value in the unsorted part of the array is swapped with the value in the `Current` location. After the swap, `Current` is in the sorted part of the array, so we shrink the size of the unsorted part by incrementing `Current`. The loop body is now complete.

Back at the top of the loop body, the unsorted part of the array goes from the (now incremented) `Current` index to the end of the array. We know that every value in the unsorted part is greater than (or equal to, if duplicates are permitted) any value in the sorted part of the array.

How do we know when there is only one element left in the unsorted part of the array? When `Current < Info'Last`, the unsorted part of the array, contains more than one

value. In each iteration of the loop body, `Current` is incremented, shrinking the unsorted part of the array. When `Current = Info'Last`, the unsorted part contains one element, and we know that this value is greater than (or equal to) any value in the sorted part. So the value in `Info(Info'Last)` is in its correct place, and we are done.

Because our selection sort has no preconditions it must be capable of sorting a null array (an array with no elements). Because a null array does not contain any elements, the loop exit condition in our algorithm can never be met; there can never be *one* element in the unsorted part. It seems obvious that we *also* want to exit our loop when there are no elements in the unsorted part of the array. Combining these conditions gives us

```
-- Exit when zero or one element left in the unsorted part of the array
exit when Current >= Info'Last;
```

A snapshot of the selection sort algorithm is illustrated in Figure 14-2.

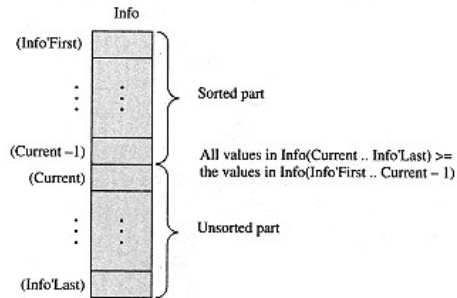


Figure 14-2  
The Straight Selection Sort

Now all we have to do is to locate the smallest value in the unsorted part of the array. Let's write a function to do this task. Function `Min_Index` is given an unconstrained array of elements and returns the index of the smallest value in this array.

#### **Min\_Index**

```
Min := Unsorted'First
for Index in Unsorted'First + 1 .. Unsorted'Last loop
 if Unsorted(Index) < Unsorted(Min) then
 Min := Index
 end if
end loop
return Min
```

Trace this algorithm on the data in Figure 14-1(a). Did you come up with a result of 4?

Now that we know where the smallest unsorted element is, we swap it with the element at index `Current`. We can use the same `Swap` procedure we used in the heap package body we developed in Chapter 13. Body 14-1 contains the code for procedure `Selection_Sort`.

**Body 14-1—****Selection Sort Implements Specification 14-1**

```

procedure Selection_Sort (Info : in out Array_Type) is

 Current : Positive; -- Index that separates sorted from unsorted parts
 Smallest : Positive; -- Index of smallest element in the unsorted part

 -- Local subprograms

 procedure Swap (Left : in out Element_Type; Right : in out Element_Type) is
 -- Swaps the values of two variables
 Temp : Element_Type;
 begin
 Temp := Left;
 Left := Right;
 Right := Temp;
 end Swap;

 function Min_Index (Unsorted : in Array_Type) return Positive is
 Min : Positive;
 begin
 Min := Unsorted'First;
 for Index in Unsorted'First + 1 .. Unsorted'Last loop
 if Unsorted(Index) < Unsorted(Min) then
 Min := Index;
 end if;
 end loop;
 return Min;
 end Min_Index;

 -- This statement is discussed later in the section "In-Line Subprograms".
 -- It makes the sort run faster by removing the overhead of subprogram calls.
 pragma INLINE (Swap, Min_Index);

begin -- Selection_Sort
 Current := Info'First;
 loop -- Each iteration, one element is moved to the correct spot
 exit when Current >= Info'Last; -- Exit when 0 or 1 elements remain
 -- Find the index of the smallest unsorted element
 Smallest := Min_Index (Unsorted => Info(Current..Info'Last));
 -- Put the smallest element into the correct spot
 Swap (Info(Current), Info(Smallest));
 -- Shrink the unsorted part of the array
 Current := Current + 1;
 end loop;
end Selection_Sort;

```



## Analyzing Selection Sort

Now let's try measuring the amount of work required by this algorithm. We describe the number of comparisons as a function of the number of elements in the array. To be concise, in this discussion we refer to `Info'Length` as  $N$ .

The comparison operation is in the `Min_Index` function. We know from the loop exit condition, `exit when Current >= Info'Last`, that `Min_Index` is called  $N - 1$  times. Within the function, the number of comparisons varies, depending on the size of the unsorted array parameter `Unsorted`:

```
for Index in Unsorted'First + 1 .. Unsorted'Last loop
 if Unsorted(Index) < Unsorted Min) then
 Min := Index;
 end if;
end loop;
```

In the first call to `Min_Index`, `Unsorted'First` is 1 and `Unsorted'Last` is `Info'Last`, so there are  $N - 1$  comparisons; in the next call there are  $N - 2$  comparisons, and so on, until in the last call there is only one comparison. The total number of comparisons is

$$(N - 1) + (N - 2) + (N - 3) + \cdots + 1 = N(N - 1)/2$$

To accomplish our goal of sorting an array of  $N$  elements, the straight selection sort requires  $N(N - 1)/2$  comparisons. Note that the particular arrangement of values in the array does not affect the amount of work done at all. Even if the array is in sorted order *before* the call to `Selection_Sort`, the procedure still makes  $N(N - 1)/2$  comparisons. The following table shows the number of comparisons required for arrays of various sizes. Note that doubling the array size roughly quadruples the number of comparisons.

**Number of Comparisons Required to Sort Arrays of Different Sizes Using Selection\_Sort**

| <i>Number of Elements</i> | <i>Number of Comparisons</i> |
|---------------------------|------------------------------|
| 10                        | 45                           |
| 20                        | 190                          |
| 100                       | 4,950                        |
| 1,000                     | 499,500                      |
| 10,000                    | 49,995,000                   |

How do we describe this algorithm in terms of Big-O? If we express  $N(N - 1)/2$  as  $1/2 N^2 - 1/2 N$ , it is easy to see. In Big-O notation we only consider the term  $1/2 N^2$ , because it increases fastest relative to  $N$ . (Remember the elephants and goldfish?) Further, we ignore the constant,  $1/2$ , making these algorithms  $O(N^2)$ . This means that, for large values of  $N$ , the computation time is *approximately proportional* to  $N^2$ . Looking back at the previous table, we see that multiplying the number of elements by 10 increases the number of comparisons by more than a factor of 100. That is, the number of comparisons is multiplied by approximately the square of the increase in the number of elements. Looking at this

chart makes us appreciate why sorting algorithms is the subject of so much attention. Using `Selection_Sort` to sort a list of 1000 elements requires almost a half million comparisons!

### Bubble Sort

The bubble sort is a selection sort that uses a different scheme for finding the minimum value. Each iteration puts the smallest unsorted element in its correct place, but it also makes changes in the location of the other elements in the array. The first iteration puts the smallest element in the array in the first array position. Starting with the last array element we compare successive pairs of elements, swapping whenever the bottom element of the pair is smaller than the one above it. In this way the smallest element "bubbles" up to the top of the array. The next iteration puts the smallest element in the unsorted part of the array into the second array position, using the same technique. As we walk through sorting the five-element array in Figure 14-3(a), note that in addition to putting one element in its proper place, each iteration causes some intermediate changes in the array.

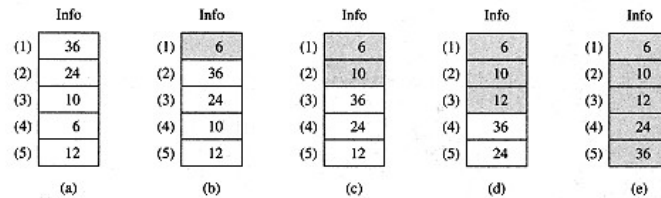


Figure 14-3  
Example of Bubble Sort

The first iteration puts 6 in `Info(1)` [Figure 14-3(b)]. Unlike the straight selection sort, which would make only the single swap between 6 and 36, the bubble sort causes two additional intermediate swaps. It first compares the values in `Info(5)` and `Info(4)`. Because  $12 \geq 6$ , no swap occurs. Then `Info(4)` and `Info(3)` are compared;  $6 < 10$ , so the two are swapped. The new value in `Info(3)`, which is 6, is compared to `Info(2)`;  $6 < 24$ , so they are swapped. Finally, the new value in `Info(2)` (6 again) is compared to `Info(1)`;  $6 < 36$ , so a swap occurs. Now the smallest value (6) is in the top position in the array. The second iteration puts the next smallest value (10) in the second position, causing one additional swap [Figure 14-3(c)], and so on.

The basic algorithm for the bubble sort is

### Bubble Sort

```
Current := Info'First
loop
 exit when the unsorted part of the array contains zero or one element
 "Bubble up" the smallest element in the unsorted part of the array,
 causing intermediate swaps as necessary
 Shrink the unsorted part of the array by incrementing Current
end loop
```

The structure of the loop is much like that of `Selection_Sort`. The unsorted part of the array is the area from `Info (Current)` to `Info (Info'Last)`. `Current` begins at `Info'First`, and we loop until `Current` reaches `Info'Last`, with `Current` incremented in each iteration. On entrance to each iteration of the loop body, the values in `Info Info'First..Current - 1` are already sorted, and all the elements in the unsorted part of the array are greater than or equal to the sorted elements.

The inside of the loop body is different, however. Each iteration of the loop "bubbles up" the smallest values in the unsorted part of the array to the `Current` position. The algorithm for the bubbling task is

### Bubble Up

```
for Index in reverse Unsorted'First + 1 .. Unsorted'Last loop
 if Unsorted(Index) < Unsorted(Index - 1) then
 Swap Unsorted(Index) with Unsorted(Index - 1)
 end if
end loop
```

A snapshot of this algorithm is shown in Figure 14-4. The code for procedure `Bubble_Sort` is given in Body 14-2.

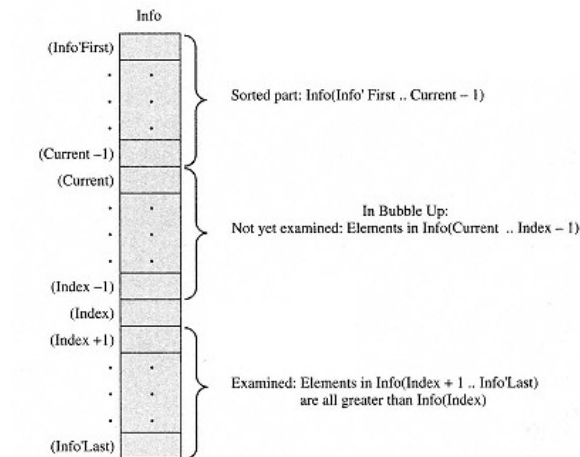


Figure 14-4  
Algorithm for the Bubble Sort

**Body 14-2—****Bubble Sort Implements Specification 14-1**

```

procedure Bubble_Sort (Info : in out Array_Type) is

 Current : Positive; -- Index that separates sorted from unsorted parts

 -- Local subprograms

 procedure Swap (Left : in out Element_Type; Right : in out Element_Type) is
 -- Swaps the values of two variables
 Temp : Element_Type;
 begin
 Temp := Left;
 Left := Right;
 Right := Temp;
 end Swap;

 procedure Bubble_Up (Unsorted : in out Array_Type) is
 begin
 for Index in reverse Unsorted'First + 1 .. Unsorted'Last loop
 if Unsorted(Index) < Unsorted(Index - 1) then
 Swap (Unsorted(Index), Unsorted(Index - 1));
 end if;
 end loop;
 end Bubble_Up;

 -- This statement is discussed later in the section "In-Line Subprograms".
 -- It makes the sort run faster by removing the overhead of subprogram calls.
 pragma INLINE (Swap, Bubble_Up);

begin -- Bubble_Sort
 Current := Info'First;
 loop -- Each iteration, one element is bubbled up to the correct spot
 exit when Current >= Info'Last; -- Exit when 0 or 1 elements remain
 -- Bubble up the smallest unsorted element
 Bubble_Up (Unsorted => Info(Current..Info'Last));
 -- Shrink the unsorted part of the array
 Current := Current + 1;
 end loop;
end Bubble_Sort;

```

**Analyzing Bubble Sort**

To analyze the work required by Bubble\_Sort is easy. It's the same as for the straight selection sort algorithm. The comparisons are in Bubble\_Up, which is called  $N - 1$  times. There are  $N - 1$  comparisons the first time,  $N - 2$  comparisons the second time, and so on. Therefore Bubble\_Sort and Selection\_Sort require the same amount of work, in terms of the number of comparisons. Bubble\_Sort does more

than just make comparisons, though; `Selection_Sort` has only one data swap per iteration, but `Bubble_Sort` may do many additional data swaps.

What is the purpose of these intermediate data swaps? By reversing out-of-order pairs of data as they are noticed, the procedure might get the array in order before  $N - 1$  calls to `Bubble_Up`. However, this version of the bubble sort makes no provision for stopping when the array is completely sorted. Even if the array is already in sorted order when `Bubble_Sort` is called, this procedure continues to call `Bubble_Up` (which changes nothing)  $N - 1$  times.

We could quit before the maximum number of iterations if `Bubble_Up` returns a Boolean flag, `Sorted`, to tell us when the array is sorted. Within `Bubble_Up`, we set `Sorted` to True; then in the loop, if any swaps are made, we reset `Sorted` to False. If no elements have been swapped, we know that the array is already in order. Now the bubble sort only needs to make *one* extra call to `Bubble_Up` when the array is in order. This version of the bubble sort is shown in Body 14-3.

### Body 14-3—

#### Short Bubble Sort Implements Specification 14-1

```
procedure Short_Bubble_Sort (Info : in out Array_Type) is

-- Stops when the array is found to be in order

 Current : Positive; -- Index that separates sorted from unsorted parts
 Sorted : Boolean;

 -- Local subprograms

 procedure Swap (Left : in out Element_Type; Right : in out Element_Type) is
 -- Swaps the values of two variables
 Temp : Element_Type;
 begin
 Temp := Left;
 Left := Right;
 Right := Temp;
 end Swap;

 procedure Bubble_Up (Unsorted : in out Array_Type;
 Sorted : out Boolean) is
 begin
 Sorted := True; -- Start out optimistic
 for Index in reverse Unsorted'First + 1 .. Unsorted'Last loop
 if Unsorted(Index) < Unsorted(Index - 1) then
 Swap (Unsorted(Index), Unsorted(Index - 1));
 Sorted := False; -- Found an element out of order
 end if;
 end loop;
 end Bubble_Up;
end Short_Bubble_Sort;
```

```

-- This statement is discussed later in the section "In-Line Subprograms".
-- It makes the sort run faster by removing the overhead of subprogram calls.
pragma INLINE (Swap, Bubble_Up);

begin -- Short_Bubble_Sort
 Current := Info'First;
 Sorted := False;
 loop -- Each iteration, one element is bubbled up to the correct spot
 exit when Sorted or Current >= Info'Last;
 -- Bubble up the smallest unsorted element
 Bubble_Up (Unsorted => Info(Current..Info'Last),
 Sorted => Sorted);
 -- Shrink the unsorted part of the array
 Current := Current + 1;
 end loop;
end Short_Bubble_Sort;

```

The analysis of Short\_Bubble\_Sort is more difficult. Clearly, if the array is already sorted, the first call to Bubble\_Up tells us so. In this best case scenario, Short\_Bubble\_Sort is  $O(N)$ ; only  $N - 1$  comparisons are required for the sort. What if the original array is actually sorted in *descending* order before the call to Short\_Bubble\_Sort? This is the worst possible case: Short\_Bubble\_Sort requires as many comparisons as Bubble\_Sort and Selection\_Sort, not to mention the overhead—all the extra swaps and setting and resetting the Sorted flag. Can we calculate an average case? In the first call to Bubble\_Up, when Current is 1, there are Info'Last - 1 comparisons; on the second call, when Current is 2, there are Info'Last - 2 comparisons. The number of comparisons in any call to Bubble\_Up is Info'Last - Current. If we let  $N$  indicate Info'Last and  $K$  indicate the number of calls to Bubble\_Up executed before Short\_Bubble\_Sort finishes its work, the total number of comparisons required is

$$\begin{array}{cccc}
 (N - 1) & + & (N - 2) & + \dots + (N - K) \\
 \text{1st call} & & \text{2nd call} & \text{3rd call} \qquad \qquad \text{Kth call}
 \end{array}$$

A little algebra\* changes this to

$$\frac{(2KN - K^2 - K)}{2}$$

In Big-O notation, the term that is increasing the fastest relative to  $N$  is  $2KN$ . We know that  $K$  is between 1 and  $N - 1$ . On the average, over all possible input orders,  $K$  is proportional to  $N$ . Therefore  $2KN$  is proportional to  $N^2$ ; that is, the short bubble sort algorithm is also  $O(N^2)$ .

---

\*For those of you who want to see the algebra:  
 $= (N - 1) + (N - 2) + \dots + (N - K)$   
 $= (KN - (\text{sum of 1 through } K))$   
 $= (KN - (1/2 K(K + 1)))$  [as we saw in Chapter 1]  
 $= (KN - (1/2 K^2 + 1/2 K))$   
 $= (2KN - K^2 - K)/2$

### Insertion Sort

In Chapter 8 we created a sorted list by inserting each new element into its appropriate place in an array. We can use a similar approach for sorting an array. As with the previous sorts, we divide our array into a sorted part and an unsorted part. To sort our array, we repeatedly take the first value out of the unsorted part and insert it at the appropriate place in the sorted part. Figure 14-5 illustrates this process which we describe in the following algorithm:

### Insertion Sort

```
Current := Info'First + 1
loop
 exit when there are no elements left in the unsorted portion
 Search the sorted portion of the array for the insertion Location of the Current element
 Make room for the Current value by moving array elements down.
 Insert the Current value into Info(Location)
 Shrink the unsorted part of the list by incrementing Current
end loop
```

Compare this algorithm to the one we developed for the Insert operation of our sequential implementation of a sorted list package in Chapter 8.

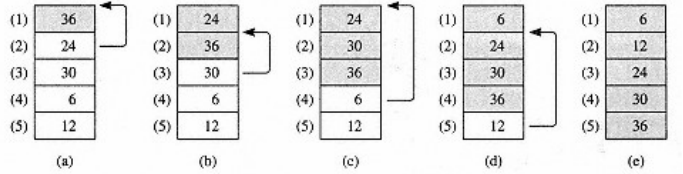


Figure 14-5  
Example of Insertion Sort

We can search the sorted part of the array with a binary search or a sequential search. For simplicity, we choose to use a sequential search. Exercise 20 asks you to implement an insertion sort with a binary search. In all our previous sequential searches, we started with the first element of our sorted array. This time there is reason to start our search at the end. Searching in the reverse direction is faster when the array passed to insertion sort is already in order. Let's demonstrate this with an example. Figure 14-6 shows the third iteration of the insertion loop for an array whose values were already in order. We are looking for the position to insert the current element, 30. Searching from beginning to end requires us to go through the entire sorted array before discovering that 30 belongs after the last element in the sorted part of the array. By starting our search at the end, we find the location immediately.

In Chapter 8 we described two methods for moving a block of array elements. We can move them with a loop or, using array slices, with a single assignment statement. We prefer the assignment method because array slice assignment usually executes much faster than a loop.\*

---

\*Most Ada compilers translate array or array slice assignment into a single machine language instruction that moves a block of memory.

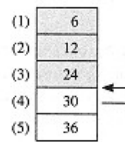


Figure 14-6  
The Insert  
Location Is Always  
After the Last  
Element in a  
Sorted Array

There are two other minor details that we must address. Our algorithm initializes `Current` to the index of the second array element. As in the selection and bubble sorts, `Current` marks the first element of the unsorted part of the array. In those sorts we started with an empty sorted part. Here we begin with a sorted part containing one element. A list containing one element is sorted. Moving the elements down destroys the current value. For example, in Figure 14-5(d) we have determined that the current element, 12 belongs between 6 and 24. Moving elements 36, 30, and 24 down overwrites 12 with 36. Therefore before we move the elements down, we make a copy of the current element. Body 14-4 contains the complete insertion sort procedure.

#### Body 14-4—

##### Insertion Sort Implements Specification 14-1

```
procedure Insertion_Sort (Info : in out Array_Type) is

 Current : Positive; -- Index that separates sorted from unsorted parts
 Location : Positive; -- Insert location
 Value : Element_Type; -- Copy of Info(Current)

 -- Local subprogram

 function Insert_Location (Sorted : in Array_Type;
 Value : in Element_Type) return Positive is
 -- Finds the location in Sorted where we should insert Value

 Index : Natural;

 begin
 Index := Sorted'Last; -- Search from end to beginning
 loop
 exit when Index < Sorted'First or else Sorted(Index) < Value;
 Index := Index - 1;
 end loop;
 return Index - 1;
 end Insert_Location;

 -- This statement is discussed later in the section "In-Line Subprograms"
 -- It makes the sort run faster by removing the overhead of subprogram calls.
 pragma INLINE (Insert_Location);
```



```

begin -- Insertion_Sort
 Current := Info'First + 1; -- Sorted part has one element at start
 loop -- Each iteration, the first element in the unsorted part of the
 -- array is inserted into the sorted part of the array
 exit when Current > Info'Last; -- Exit when no elements remain in the unsorted part
 -- Find where in the sorted portion of the array to insert the current element
 Location := Insert_Location (Sorted => Info(Info'First .. Current - 1),
 Value => Info(Current));
 -- Make a copy of Info(Current)
 Value := Info(Current);
 -- Open up a space for the element by sliding all below it down
 Info(Location + 1 .. Current) := Info(Location .. Current - 1);
 -- Insert the current value
 Info(Location) := Value;
 -- Shrink the unsorted part of the array
 Current := Current + 1;
 end loop;
end Insertion_Sort;

```

### Analysis of Insertion Sort

All of the comparisons in this sort are made in the search procedure. If the array elements are already ordered, each call to the search makes only one comparison. Because we call the search procedure  $N - 1$  times, the number of comparisons for sorting an array whose elements are already in order is  $N - 1$ . Insertion sort is very efficient when the array is already sorted.

Now let's look at the worst case—one comparison is made the first time the search is called, two comparisons the second time, three comparisons the third time, and so on.  $N - 1$  comparisons are made the last time the search is called. The number of comparisons in the worst case is

$$1 + 2 + 3 + \cdots + (N - 3) + (N - 2) + (N - 1) = N(N - 1)/2$$

The maximum number of comparisons is made only when the elements in the array are in reverse order. On the average, for a randomly ordered array, we search just half of the sorted part before finding the insert location. Therefore for a randomly ordered array the total number of comparisons is closer to  $N(N - 1)/4$ . Because this value is half the number of comparisons made by selection sort and bubble sort, we might expect insertion sort to be about twice as fast in sorting an array of mixed-up values. The timing tests given in the summary of this chapter support this observation.

Even though insertion sort may be faster than the selection or bubble sorts, all are on the order of  $N^2$ ; they are all too time-consuming for sorting *large* arrays. Thus there's a need for sorting methods that work better when  $N$  is large.

### $O(N \log_2 N)$ Sorts

Considering how fast  $N^2$  grows as the size of the array gets larger, can't we do better? We note that  $N^2$  is a lot larger than  $(1/2 N)^2 + (1/2 N)^2$ . If we could cut the array into two pieces,

sort each segment, and then merge the two back together, we should end up sorting the entire array with a lot less work. An example of this approach is shown in Figure 14-7.

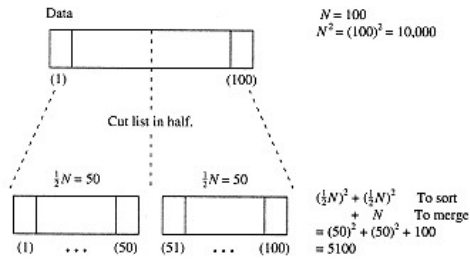


Figure 14-7  
Rationale for Divide-and-Conquer Sorts

The idea of "divide and conquer" has been applied to the sorting problem in different ways, resulting in a number of algorithms that can do the job much more efficiently than  $O(N^2)$ . We look at two of these sorting algorithms here; merge sort and quick sort. As you might guess, the efficiency of these algorithms is achieved at the expense of the simplicity seen in the straight selection and bubble sorts.

### **Merge Sort**

The merge sort algorithm is taken directly from the idea in the previous section:

#### **Merge Sort**

Cut the list in half  
 Sort the left half  
 Sort the right half  
 Merge the two sorted halves into one sorted list

Merging the two halves together is an  $O(N)$  task. We merely go through the sorted halves, comparing successive pairs of values (one in each half) and putting the smaller value into the next slot in the final solution. Even if the sorting algorithm used for each half is  $O(N^2)$ , we should see some improvement over sorting the whole list at once.

Actually, because merge sort is itself a sorting algorithm, we might as well use it to sort the two halves. That's right—we can make `Merge_Sort` a recursive procedure and let it call itself to sort each of the two sublists:

#### **Merge\_Sort – Recursive**

Cut the list in half  
 Merge\_Sort the left half  
 Merge\_Sort the right half  
 Merge the two sorted halves into one sorted list

This is the general case, of course. What is the base case, the case that does not involve any recursive calls to `Merge_Sort`? If the half to be sorted doesn't have more

than one element, we can consider it already sorted and just return. Here is our algorithm with this base case:

### Merge\_Sort – Recursive

```

if the array Info contains more than one element then
 Cut the array in half -- General Case
 Merge_Sort the left half of Info
 Merge_Sort the right half of Info
 Merge the two sorted halves into one sorted list
else
 Do Nothing -- Base Case
end if

```

Cutting the array in half is simply a matter of finding the midpoint between the first and last indexes:

```
Middle := (Info'First + Info'Last) / 2;
```

Then, in the smaller-caller tradition, we can make the recursive calls to Merge\_Sort:

```

Merge_Sort (Info => Info(Info'First .. Middle);
Merge_Sort (Info => Info(Middle + 1 .. Info'Last));

```

So far this is pretty simple. Now we only have to merge the two halves and we're done.

### Merging the Sorted Halves

Obviously all the serious work is in the merge step. Let's first look at the general algorithm for merging two sorted arrays, and then we can look at the specific problem of our subarrays.

To merge two sorted arrays, we compare successive pairs of elements, one from each array, moving the smaller of each pair to the "final" array. We can stop when the shorter array runs out of elements, and then move all the remaining elements (if any) from the other array to the final array. Figure 14-8 illustrates the general algorithm.

We use a similar approach in our specific problem, in which the two "arrays" to be merged are actually slices of the original array (Figure 14-9). Just as in the previous example we merged Right and Left into a third array, we need to merge our two slices into another array, temporarily. After the merge step, we copy the now-sorted elements back into the original array. The whole process is shown in Figure 14-10. Let's specify a function, Merge, to do this task:

```

function Merge (Left : in Array_Type;
 Right : in Array_Type) return Array_Type;
-- Merge two sorted arrays into a single sorted array.

-- Preconditions: Left is sorted.
-- Right is sorted.

-- Postconditions: A sorted array containing all the elements in Left
-- and Right is returned.

```

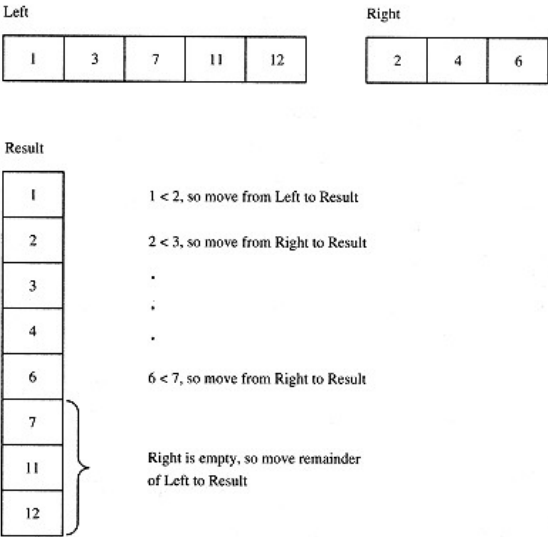


Figure 14-8  
Strategy for Merging Two Sorted Arrays

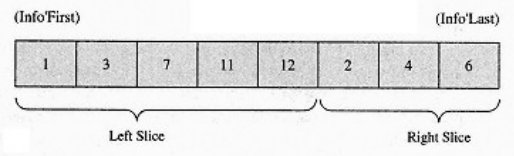


Figure 14-9  
Two Array Slices

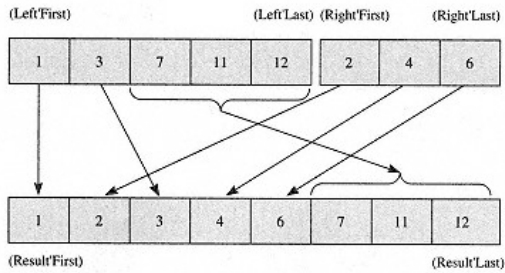


Figure 14-10  
Merging Sorted Halves

Here is the algorithm for function Merge:

### Merge

```

Left_Index := Left'First
Right_Index := Right'First
Result_Index := Result'First
loop
 exit when Left or Right run out of elements
 if Left(Left_Index) < Right(Right_Index) then
 Result(Result_Index) := Left(Left_Index)
 Increment Left_Index
 Increment Result_Index
 else
 Result(Result_Index) := Right(Right_Index)
 Increment Right_Index
 Increment Result_Index
 end if
end loop
if there are any elements remaining in Left then
 Copy remaining elements in Left to Result
else
 Copy remaining elements in Right to Result
end if
return Result

```

Because both branches of the first if statement increment Result\_Index, we can replace the two instructions with a single instruction after the if statement. We declare the local array Result so its length is equal to the sum of the lengths of Left and Right. And we can efficiently copy the remaining elements with array slices.

As we said, most of the work is in the merge task. The actual Merge\_Sort procedure (Body 14-5) is short and simple.

### Body 14-5—

#### Merge Sort Implements Specification 14-1

```

procedure Merge_Sort (Info : in out Array_Type) is

 Middle : Positive; -- Middle index of array Info

 -- Local subprogram

 function Merge (Left : in Array_Type;
 Right : in Array_Type) return Array_Type is
 Left_Index : Positive;
 Right_Index : Positive;
 Result_Index : Positive;
 Result : Array_Type (1 .. Left'Length + Right'Length);

```

```

begin -- Merge
 -- Initialize array indexes
 Left_Index := Left'First;
 Right_Index := Right'First;
 Result_Index := Result'First;
 loop -- Each iteration one element is moved into Result
 -- Exit when one array is "empty"
 exit when Left_Index > Left'Last or
 Right_Index > Right'Last;
 if Left_Index < Right_Index then
 Result(Result_Index) := Left(Left_Index); -- Copy from Left
 Left_Index := Left_Index + 1;
 else
 Result(Result_Index) := Right(Right_Index); -- Copy from Right
 Right_Index := Right_Index + 1;
 end if;
 Result_Index := Result_Index + 1;
 end loop;

 -- Copy any remaining elements
 if Left_Index <= Left'Last then -- Still elements remaining in Left?
 Result(Result_Index .. Result'Last) := Left(Left_Index .. Left'Last);
 else
 Result(Result_Index .. Result'Last) := Right(Right_Index .. Right'Last);
 end if;
 return Result;
end Merge;

begin -- Merge_Sort
 -- Base Case: Check for empty array or single element array
 if Info'Length > 1 then
 Middle := (Info'First + Info'Last) / 2; -- Divide the array in half
 Merge_Sort (Info => Info(Info'First .. Middle)); -- Sort the left half
 Merge_Sort (Info => Info(Middle + 1 .. Info'Last)); -- Sort the right half
 -- Merge the two sorted halves back into the original array
 Info := Merge (Left => Info(Info'First .. Middle),
 Right => Info(Middle + 1 .. Info'Last));
 end if;
end Merge_Sort;

```

### Analyzing Merge Sort

We already pointed out that sorting two half-lists is less work than sorting one whole list. How much less work is it? The bulk of the work occurs in the merge processing. In the `Merge` function we make comparisons on each element in the two arrays. Because the combined number of elements in the two arrays is  $N$ , this is an  $O(N)$  operation. So the `Merge` function is  $O(N)$ .

Now, how many times is the `Merge` function called? It's called in procedure `Merge_Sort` after the array has been divided in half and each of those halves has been sorted (using `Merge_Sort`, of course). In each of the recursive calls, one for the left half and one for the right, the array is divided in half again, making four pieces. Each of these

pieces is similarly subdivided. At each level the number of pieces doubles (see Figure 14-11). We can keep dividing the array in half  $\log_2 N$  times. (This is just like the binary search algorithm in Chapter 2.)

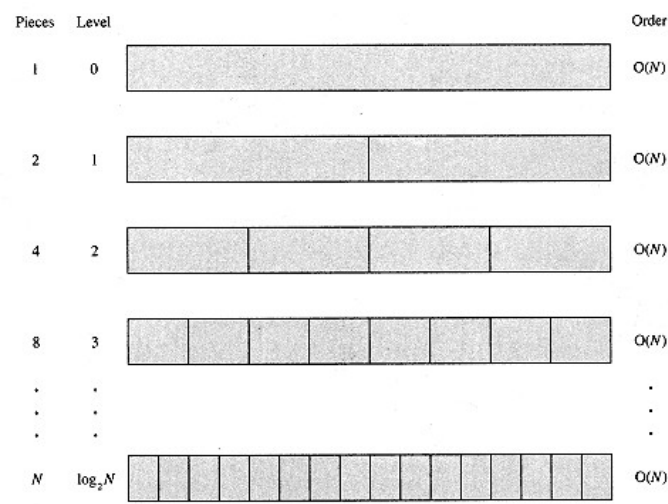


Figure 14-11  
Analysis of Procedure Merge\_Sort

Each time the array is divided, we perform the  $O(N)$  Merge function to put it back together again. This gives us a product of  $N \times \log_2 N$ . Thus the whole algorithm is  $O(N \log_2 N)$ . Figure 14-12 illustrates that, for large values of  $N$ ,  $O(N \log_2 N)$  is a big improvement over  $O(N^2)$ .

| $N$  | $\log_2 N$ | $N^2$      | $N \log_2 N$ |
|------|------------|------------|--------------|
| 32   | 5          | 1,024      | 160          |
| 64   | 6          | 4,096      | 384          |
| 128  | 7          | 16,384     | 896          |
| 256  | 8          | 65,536     | 2,048        |
| 512  | 9          | 262,144    | 4,608        |
| 1024 | 10         | 1,048,576  | 10,240       |
| 2048 | 11         | 4,194,304  | 22,528       |
| 4096 | 12         | 16,777,216 | 49,152       |

Figure 14-12  
Comparing  $N^2$  and  $N \log_2 N$

The disadvantage of Merge\_Sort is that it requires an auxiliary array (the local variable Result) that, in the last merge, is as large as the original array to be sorted. If the array is large and space is a critical factor, this sort may not be an appropriate choice. Next we discuss two sorts that move elements around in the original array and do not need an auxiliary array.

## Quick Sort

The quick sort is another sorting algorithm based on the fact that it's faster and easier to sort two small lists than one large one. As the name indicates, in general quick sort can sort a list of data elements significantly faster than any of the common simple sorts. The basic strategy of quick sort, like that of merge sort, is to divide and conquer.

If you were given a large stack of final exams to sort by name, you might use the following approach: Pick a splitting value—L, for example; divide the stack of tests into two piles, A–L and M–Z. (Note that the two piles may not necessarily contain the same number of tests.) Then take the first pile and subdivide it into two piles, A–F and G–L. The A–F pile can be further broken down into A–C and D–F. This division process goes on until the piles are small enough to be easily sorted. The same process is applied to the M–Z pile.

Eventually all the small sorted piles can be stacked one on top of the other to produce an ordered set of tests. (See Figure 14-13.)

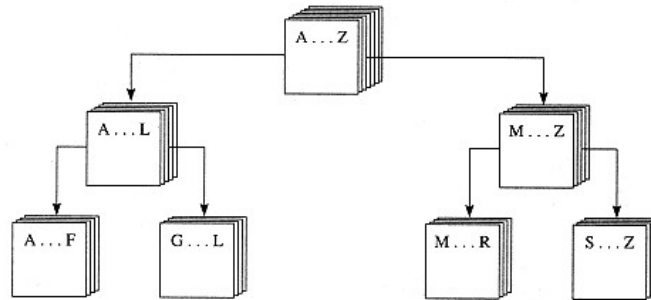


Figure 14-13  
Ordering Exams Using Quick Sort

This strategy is based on recursion—on each attempt to sort the stack of tests the stack is divided and then the same approach is used to sort each of the smaller stacks (a smaller case). This process goes on until the small stacks do not need to be further divided (the base case). Here is the top level algorithm for quick sort:

### Quick Sort

```

if Info contains more than one element then
 Split Info into two portions
 Quick Sort the left portion
 Quick Sort the right portion
end if

```

An obvious difference between the quick sort algorithm and the merge sort algorithm is the absence of a step to combine the two sorted portions—the merge. Because some sorting is done by each split (see Figure 14-13), the elements are already in their proper positions when we reach the base case of the recursive call.

### Splitting the Array: The Abstract Level

Splitting the array is the easy part of the merge sort; we simply split the array in half. In quick sort, we split the array so that all of the



elements in the left portion are less than all of the elements in the right portion. Separating these two portions is an element we call the *split value*. Some programmers call it the *pivot value*. More formally the elements in `Info` satisfy the following assertions:

- All elements in `Info (Info'First .. Split_Index - 1) < Split Value`
- `Info (Split_Index) = Split Value`
- All elements in `Info (Split_Index + 1 .. Info'Last) >= Split Value`

How do we select the split value? One simple solution is to use the first value in the array as the splitting value [Figure 14-14(a)]. The `Split` procedure moves the split value to the correct position in the array (`Split_Index`) and moves some of the other elements so that all the elements less than the split value are on the left side of the array, and all those greater than or equal to the split value are on the right [Figure 14-14(b)]. Note that we don't know the value of `Split_Index` until the splitting process is complete.

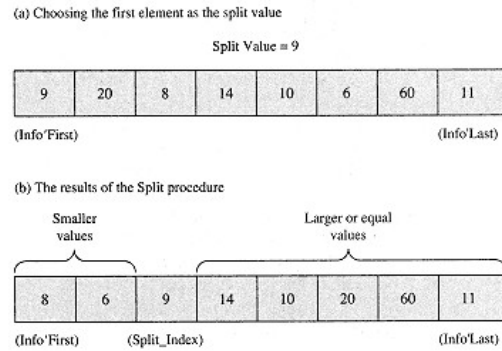


Figure 14-14  
Results of Procedure Split

Our recursive calls to `Quick_Sort` use `Split_Index` to reduce the size of the problem in the general case. The left portion of the array is sorted by the call:

```
Quick_Sort (Info => Info(Info'First .. Split_Index - 1));
```

and the right portion of the array is sorted by the call:

```
Quick_Sort (Info => Info(Split_Index + 1 .. Info'Last));
```

Let's verify `Quick_Sort` according to the Three-Question Method.

1. *Is there a nonrecursive base case?* Yes. When `Info'Length <= 1`, `Quick_Sort` does nothing.
2. *Does each recursive call involve a smaller case of the problem?* Yes. `Split` divides the segment into two not necessarily equal pieces, and each of these smaller pieces is then quick sorted. Note that even if the split value is the largest or smallest value in the segment, the two pieces are still smaller than the original one: If the split value is smaller than *all* the other values in the segment, then the left portion of the array

`Info(Info'First..Split_Index - 1)` is a null array, and the right portion `Info(Split_Index + 1..Info'Last)` contains one less element (the split value) than the original array.

3. Assuming that the recursive calls succeed, does the whole procedure work? Yes. We assume that `Quick_Sort(Info(Info'First..Split_Index - 1))` actually sorts the first `Split_Index - 1` elements, whose values are less than the split value. `Info(Split_Index)`, containing the split value, is in its correct place. We also assume that `Quick_Sort(Info(Split_Index + 1..Info'Last))` has correctly sorted the rest of the list, whose values are all greater than the split value. So we know that the whole list is sorted.

### Splitting the Array:

#### The Implementation Level

In good top-down fashion, we have shown that our algorithm works if procedure `Split` works. Now we must develop our splitting algorithm. We must find a way to get all of the elements less than the split value on one side of `Split_Index` and the elements equal to or greater than the split value on the other side.

We do this by moving two indexes, `Left` and `Right`, toward the middle of the array from the ends, looking for elements that are on the wrong side of the split point (Figure 14-15). Initially `Left` identifies the second element in the array and `Right` identifies the last element in the array. [See Figure 14-15(a).]

We start out by moving `Left` toward the middle, comparing `Info(Left)` to the split value. If `Info(Left) < the split value`, we keep incrementing `Left`; otherwise, we leave `Left` where it is and begin moving `Right` toward the middle. [See Figure 14-15(b).]

Now `Info(Right)` is compared to the split value. If it is greater than or equal to the split value, we continue decrementing `Right`; otherwise, we leave `Right` in place. [See Figure 14-15(c).] At this point it's clear that `Info(Left)` and `Info(Right)` are each on the wrong side of the array. Note that the elements to the left of `Info(Left)` and to the right of `Info(Right)` are not necessarily sorted; they are just on the correct side *with respect to the split value*. To put `Info(Left)` and `Info(Right)` into their correct sides, we merely swap them, then increment `Left` and decrement `Right`. [See Figure 14-15(d).]

Now we repeat the whole cycle, incrementing `Left` until we encounter a value that is greater than or equal to the split value, and then decrementing `Right` until we encounter a value that is less than the split value. [See Figure 14-15(e).]

When does the process stop? When `Left` and `Right` meet each other. They meet at the `Split_Index`. This location is where the split value belongs, so we swap `Info(Info'First)`, which contains the split value, with the element at `Info(Split_Index)` [Figure 14-15(f)]. `Split_Index` is returned from the procedure, to be used by `Quick_Sort` to set up the next recursive call.

Body 14-6 (page 678) contains our code that implements the quick sort algorithm. We recommend that you spend some time walking through the `Split` procedure with different sized arrays containing values in different orders. Do you see why we initialized `Left` to `Info'First + 1` rather than `Info'First`? And do you see what special case requires us to use `or else` rather than `or` in the exit statements of the first inner loop?

Our quick sort algorithm uses the `">="` operator to compare array elements. However, the application programmer supplies only the `"<"` operator as a generic formal function. Rather than complicate the code of `Split` with the necessary Boolean transformations or

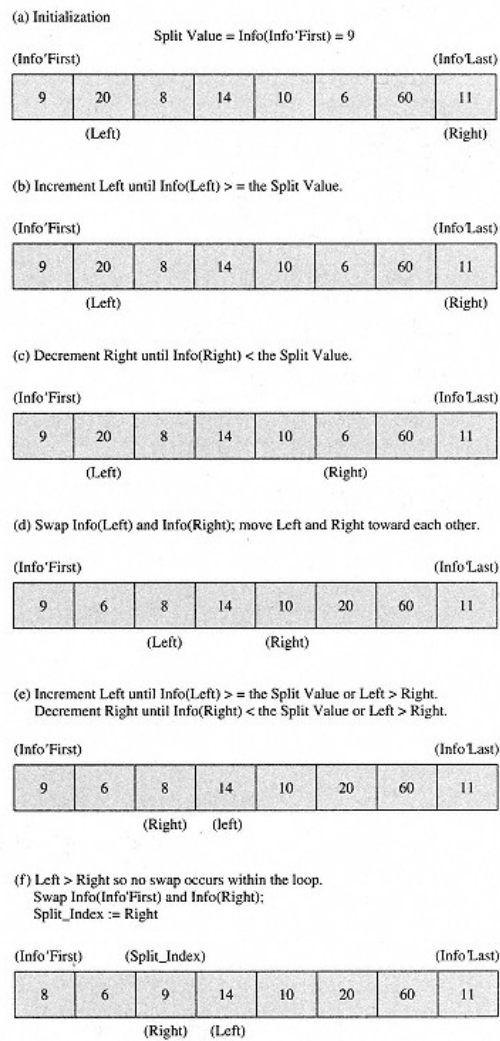


Figure 14-15  
Trace of Procedure Split

add another generic formal function to our generic declaration, we use the application-programmer-supplied function to write our own ">=" operator.

What happens if our splitting value is the largest or the smallest value in the segment? The algorithm still works correctly, but because of the lopsided splits it isn't quick (we discuss the performance of quick sort in the next section).

Is this situation likely to occur? That depends on how we choose our splitting value and on the original order of the data in the array. If we use `Info (Info'First)` as the splitting value and the array is already sorted, then *every* split is lopsided. One side contains no elements, while the other side contains all but one of the elements (the split value). Thus our `Quick_Sort` is not a quick sort. This splitting algorithm favors an array in random order.

It's not unusual, however, to want to sort an array that is already in *nearly* sorted order. If this is the case, a better splitting value would be the middle value.

```
Info ((Info'First + Info'Last) / 2)
```

This value can be swapped with `Info (Info'First)` at the beginning of the `Split` procedure.

There are many other splitting algorithms. We have only discussed the most basic. You'll probably encounter others in future courses, other books, computer science journals, or in someone else's programs.

#### Body 14-6—

#### Quick Sort Implements Specification 14-1

```
procedure Quick_Sort (Info : in out Array_Type) is

 Split_Index : Positive;

 procedure Swap (Left : in out Element_Type; Right : in out Element_Type) is
 -- Swaps the values of two variables
 Temp : Element_Type;
 begin
 Temp := Left;
 Left := Right;
 Right := Temp;
 end Swap;

 function ">=" (Left : in Element_Type; Right : in Element_Type) return Boolean is
 begin
 return not (Left < Right);
 end ">=";

 procedure Split (Info : in out Array_Type;
 Split_Index : out Positive) is
 Left : Positive;
 Right : Positive;
```

```

begin -- Split
 Left := Info'First + 1;
 Right := Info'Last;
 loop -- Each iteration, two out of place values are swapped
 -- Left to right search
 loop -- Each iteration one element is inspected
 -- Exit when searches cross or find an element on wrong side
 exit when Left > Right or else Info(Left) >= Info(Info'First);
 Left := Left + 1;
 end loop;
 -- Right to left search
 loop -- Each iteration one element is inspected
 -- Exit when searches cross or find an element on wrong side
 exit when Left > Right or Info(Right) < Info(Info'First);
 Right := Right - 1;
 end loop;
 -- Exit outer loop when searches cross
 exit when Left > Right;
 -- Swap the two values on the wrong side of the split value
 Swap (Info Left), Info(Right));
 Left := Left + 1;
 Right := Right - 1;
 end loop;
 -- Move the split value into its proper location
 Swap (Info(Info'First), Info(Right));
 -- Return the index of the value that divides the two portions
 Split_Index := Right;
end Split;

-- This statement is discussed later in the section "In-Line Subprograms".
-- It makes the sort run faster by removing the overhead of subprogram calls.
pragma INLINE (Swap, ">=");

begin -- Quick_Sort
 if Info'Length > 1 then
 Split (Info, Split_Index);
 Quick_Sort (Info => Info(Info'First .. Split_Index - 1));
 Quick_Sort (Info => Info(Split_Index + 1 .. Info'Last));
 end if;
end Quick_Sort;

```

### Analyzing Quick Sort

The analysis of `Quick_Sort` is very similar to that of `Merge_Sort`. On the first call, every element in the list is compared to the dividing value, so the work done is  $O(N)$ . The array is divided into two parts (not necessarily halves), which are then examined.

Each of these pieces is then divided in two, and so on. If each piece is split approximately in half, there are  $O(\log_2 N)$  splits. At each split, we make  $O(N)$  comparisons. So `Quick_Sort` is also an  $O(N \log_2 N)$  algorithm, which is quicker than the  $O(N^2)$  sorts we discussed at the beginning of this chapter.

But `Quick_Sort` isn't *always* quicker. Note that there are  $\log_2 N$  splits only if each split divides the segment of the array approximately in half. As we've seen, `Quick_Sort` is sensitive to the order of the data.

What happens if the array is already sorted when `Quick_Sort` is called? If we use the first element as the split value, the splits are very lopsided, and the subsequent recursive calls to `Quick_Sort` sort a segment of no elements and a segment containing all the elements except the split value. This situation produces a sort that is not at all quick. In fact, there will be  $N - 1$  splits; in this case `Quick_Sort` is  $O(N^2)$ .

It's not uncommon for data to be nearly sorted. In these situations, using the middle element as the split value gives much better performance.

Why would we even consider using `Quick_Sort` when it's sometimes much slower than `Merge_Sort`? As we discussed in our analysis of it, `Merge_Sort` requires enough memory to hold two copies of the data. If space is a concern, `Quick_Sort` is a good choice. `Merge_Sort` also requires a good deal of data movement. Each call to the merge function moves all the data twice: once from the two array parameters to the local array and then back to the original array when the function's result is assigned to the original array. Thus when the size of an element is large, `Quick_Sort` is usually a better choice.

### **Heap Sort**

In each iteration of the selection sort, we searched the array for the next smallest element and put it in its correct place in the array. Another way to write a selection sort is to find the maximum value in the array and swap it with the last array element, then find the next-to-largest element and put it in its place, and so on. Most of the work in this sorting algorithm comes from searching the remaining part of the array in each iteration, looking for the maximum value.

In Chapter 13 we discussed the *heap*, a data structure with a very special feature—we always know where to find its greatest element. Because of the order property of heaps, the maximum value of a heap is in the root node. We can take advantage of this situation by using a heap to help us sort. The general approach of heap sort is as follows:

1. Take the root (maximum) element off the heap, and put it in its place.
2. Reheap the remaining elements. (This puts the next-largest element back in the root position.)
3. Repeat until there are no more elements.

The first part of this algorithm sounds a lot like the straight selection sort. What makes the heap sort fast is the second step: finding the next-largest element. Because the shape property of heaps guarantees a binary tree of minimum height, we make only  $O(\log_2 N)$  comparisons in each iteration, as compared with  $O(N)$  comparisons in each iteration of the selection sort.

### **Building a Heap**

By now you are probably protesting that we are dealing with an unsorted array of elements, not a heap. Where does the original heap come from? Before we go on, we'll have to convert the unsorted array, `Info`, into a heap.

Let's take a look at how the heap relates to our array of unsorted elements. In Chapter 13 we saw how heaps can be represented in an array with implicit links. Because of the shape property, we know that the heap elements take up consecutive positions in the array. In fact, the unsorted array of data elements already satisfies the shape property of heaps. Figure 14-16 shows an unsorted array and its equivalent tree.

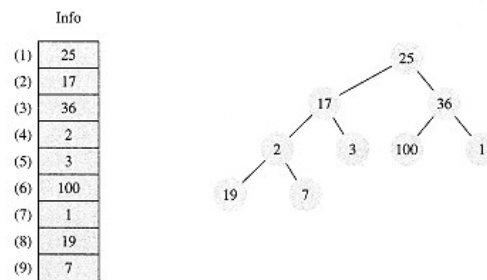


Figure 14-16  
An Unsorted Array and Its Tree

We also need to make the unsorted array elements satisfy the order property of heaps. First let's see if there's any part of the tree that already satisfies the order property. All of the leaf nodes (subtrees with only a single node) are heaps. In Figure 14-17(a) the subtrees whose roots contain the values 19, 7, 3, 100, and 1 are heaps because they are root nodes.

Now let's look at the first *nonleaf* node, the one containing the value 2 [Figure 14-17(b)]. The subtree rooted at this node is not a heap, but it is *almost* a heap—all of the nodes *except the root node* of this subtree satisfy the order property. We know how to fix this problem. In Chapter 13 we developed a heap utility procedure, `Reheap_Down`, that can be used to correct this exact situation. Given a tree whose elements satisfy the order property of heaps except (perhaps) at the root node, `Reheap_Down` rearranges the nodes, leaving the (sub)tree as a heap.

We apply this procedure to all the subtrees on this level, then we move up a level in the tree and continue reheapifying until we reach the root node. After `Reheap_Down` has been called for the root node, the whole tree should satisfy the order property of heaps. This heap-building process is illustrated in Figure 14-17; the changing contents of the array are shown in Figure 14-18.

The algorithm for this process is summarized below:

#### Build Heap

```
for Index going from first nonleaf node up to the root node loop
 Reheap_Down (Heap => Info(Index..Info'Last))
end loop
```

We know where the root node is stored in our array representation of heaps—it's in `Info(1)`. Where is the first nonleaf node? Because half the nodes of a complete binary tree are leaves (prove this yourself), the first nonleaf node may be found at `Info(Info'Last/2)`.

Because our heap storage scheme requires that the root node be at location 1, heap sort has the following precondition:

```
-- Preconditions: Info'First must be 1
```

Although our other sorts could sort an array with arbitrary starting and ending index values, the starting index of the array for heap sort must be 1.

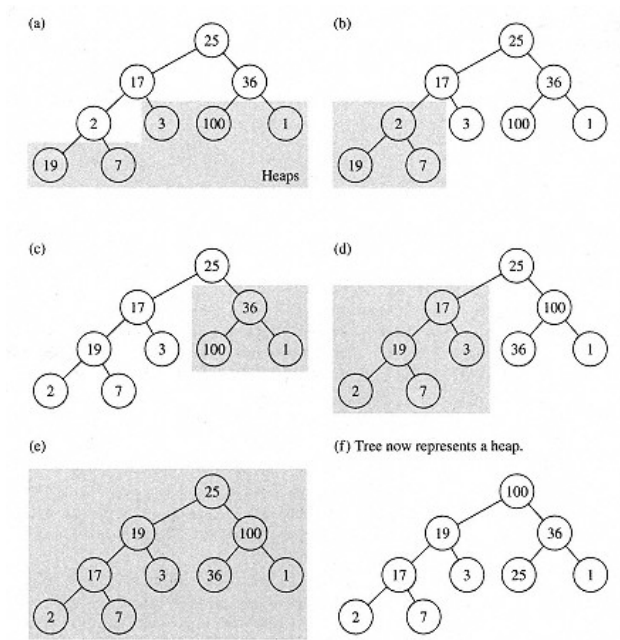


Figure 14-17  
The Heap-Building Process

|                             | (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) | (9) |
|-----------------------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Original Info               | 25  | 17  | 36  | 2   | 3   | 100 | 1   | 19  | 7   |
| After ReapDown<br>Index = 4 | 25  | 17  | 36  | 19  | 3   | 100 | 1   | 2   | 7   |
| After Index = 3             | 25  | 17  | 100 | 19  | 3   | 36  | 1   | 2   | 7   |
| After Index = 2             | 25  | 19  | 100 | 17  | 3   | 36  | 1   | 2   | 7   |
| After Index = 1             | 100 | 19  | 36  | 17  | 3   | 25  | 1   | 2   | 7   |

Tree is a heap.

Figure 14-18  
Changing Contents of the Array

### Sorting the Heap

Now that we are satisfied that we can turn the unsorted array of elements into a heap, let's take another look at the sorting algorithm.

We can easily access the largest element from the original heap—it's in the root node. In our array representation of heaps, that's `Info(1)`. This value belongs in the last-used array position `Info(Info'Last)`, so we can just swap the values in these two positions. Because `Info(Info'Last)` now contains the largest value in the array (its correct sorted value), we want to leave this position alone. Now we are dealing with a set of elements, from `Info(1)` through `Info(Info'Last - 1)`, that is almost a heap. We know



that all of these elements satisfy the order property of heaps, except (perhaps) the root node. To correct this condition, we call our heap utility, `Reheap_Down`.

At this point we know that the next-largest element in the array is in the root node of the heap. To put this element in its correct position, we swap it with the element in `Info(Info'Last - 1)`. Now the two largest elements are in their final correct positions, and the elements in `Info(1)` through `Info(Info'Last - 2)` are almost a heap. So we call `Reheap_Down` again, and now the third-largest element is in the root of the heap.

This process is repeated until all of the elements are in their correct positions; that is, until the heap contains only a single element, which must be the smallest item in the array, in `Info(1)`. This is its correct position, so the array is now completely sorted from the smallest to the largest element. Notice that at each iteration the size of the unsorted portion (represented as a heap) gets smaller and the size of the sorted portion gets larger. At the end of the algorithm, the size of the sorted portion is the size of the original array.

The heap sort algorithm, as we have described it, sounds like a recursive process. Each time we swap and reheap a smaller portion of the total array. Because it uses tail recursion, we can code the repetition just as clearly using a simple for loop. The node sorting algorithm is

### Sort Nodes

```
for Index in last node up to next-to-root node loop
 Swap Root node with Info(Index);
 Reheap_Down (Heap => Info(1..Index - 1))
end loop
```

Procedure `Heap_Sort`, given in Body 14-7, first builds the heap and then sorts the nodes, using the algorithms just discussed. Figure 14-19 shows how each iteration of the sorting loop (the second for loop) would change the heap created in Figure 14-18. Each line represents the array after one operation. The sorted elements are shaded.

### Body 14-7—

#### Heap Sort Implements Specification 14-1

```
with Heap;
procedure Heap_Sort (Info : in out Array_Type) is

 -- Functions needed to instantiate a heap
 function ">=" (Left : in Element_Type; Right : in Element_Type) return Boolean;
 function Identity (Element : in Element_Type) return Element_Type;

 -- Instantiate a heap of elements
 package Element_Heap is new Heap (Element_Type => Element_Type;
 Key_Type => Element_Type;
 Heap_Array => Array_Type;
 Key_Of => Identity;
 ">" => ">=");

 -- Local subprograms
```

```

function Identity (Element : in Element_Type) return Element_Type is
begin
 return Element;
end Identity;

function ">=" (Left : in Element_Type; Right : in Element_Type) return Boolean is
begin
 return not (Left < Right);
end ">=";

procedure Swap (Left : in out Element_Type; Right : in out Element_Type) is
-- Swaps the values of two variables
 Temp : Element_Type;
begin
 Temp := Left;
 Left := Right;
 Right := Temp;
end Swap;

-- This statement is discussed later in the section "In-Line Subprograms".
-- It makes the sort run faster by removing the overhead of subprogram calls.
pragma INLINE (Swap);

begin -- Heap_Sort
 -- Build the original heap from the unsorted elements
 for Index in reverse 1 .. Info'Last / 2 loop
 Element_Heap.Reheap_Down (Heap => Info(Index..Info'Last));
 end loop;

 -- Sort the elements in the heap by swapping the root (current largest) with
 -- the last unsorted value, then reheapifying the remaining part of the array.
 for Index in reverse 2 .. Info'Last loop
 Swap (Info(1), Info(Index));
 Element_Heap.Reheap_Down (Heap => Info(1 .. Index - 1));
 end loop;
end Heap_Sort;

```

### Analyzing Heap Sort

The code for procedure `Heap_Sort` is very short—only a few lines of new code plus the utility procedure `Reheap_Down`, which we developed in Chapter 13. These few lines of code, however, do quite a bit. All of the elements in the original array are rearranged to satisfy the order property of heaps, moving the largest element up to the top of the array, only to put it immediately into its place at the bottom. It's hard to believe from a small example such as the one in Figure 14-19 that `Heap_Sort` is really very efficient.

For small arrays, `Heap_Sort` is not very efficient because of all the overhead. For large arrays, however, `Heap_Sort` is very efficient. Let's consider the sorting loop. We loop through  $N - 1$  times, swapping elements and reheapifying. The comparisons occur in

|                        | (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) | (9) |
|------------------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Info                   | 100 | 19  | 36  | 17  | 3   | 25  | 1   | 2   | 7   |
| Swap                   | 7   | 19  | 36  | 17  | 3   | 25  | 1   | 2   | 100 |
| Reheap_Down            | 36  | 19  | 25  | 17  | 3   | 7   | 1   | 2   | 100 |
| Swap                   | 2   | 19  | 25  | 17  | 3   | 7   | 1   | 36  | 100 |
| Reheap_Down            | 25  | 19  | 7   | 17  | 3   | 2   | 1   | 36  | 100 |
| Swap                   | 1   | 19  | 7   | 17  | 3   | 2   | 25  | 36  | 100 |
| Reheap_Down            | 19  | 17  | 7   | 1   | 3   | 2   | 25  | 36  | 100 |
| Swap                   | 2   | 17  | 7   | 1   | 3   | 19  | 25  | 36  | 100 |
| Reheap_Down            | 17  | 3   | 7   | 1   | 2   | 19  | 25  | 36  | 100 |
| Swap                   | 2   | 3   | 7   | 1   | 17  | 19  | 25  | 36  | 100 |
| Reheap_Down            | 7   | 3   | 2   | 1   | 17  | 19  | 25  | 36  | 100 |
| Swap                   | 1   | 3   | 2   | 7   | 17  | 19  | 25  | 36  | 100 |
| Reheap_Down            | 3   | 1   | 2   | 7   | 17  | 19  | 25  | 36  | 100 |
| Swap                   | 2   | 1   | 3   | 7   | 17  | 19  | 25  | 36  | 100 |
| Reheap_Down            | 2   | 1   | 3   | 7   | 17  | 19  | 25  | 36  | 100 |
| Swap                   | 1   | 2   | 3   | 7   | 17  | 19  | 25  | 36  | 100 |
| Reheap_Down            | 1   | 2   | 3   | 7   | 17  | 19  | 25  | 36  | 100 |
| Exit from sorting loop | 1   | 2   | 3   | 7   | 17  | 19  | 25  | 36  | 100 |

Figure 14-19  
Effect of Heap\_Sort on the Array

**Reheap\_Down.** A complete binary tree with  $N$  nodes has  $O(\log_2(N+1))$  levels. In the worst cases, then, if the root element had to be bumped down to a leaf position, the **Reheap\_Down** procedure would make  $O(\log_2 N)$  comparisons. So procedure **Reheap\_Down** is  $O(\log_2 N)$ . Multiplying this activity by the  $N - 1$  iterations  $[O(N)]$  shows that the sorting loop is  $O(N \log_2 N)$ .

Combining the original heap build, which is  $O(N)$ , and the sorting loop, we can see that **Heap\_Sort** requires  $O(N \log_2 N)$  comparisons. Note that, unlike **Quick\_Sort**, **Heap\_Sort**'s efficiency is *not* affected by the initial order of the elements. **Heap\_Sort** is just as efficient in terms of space; only one array is used to store the data.

## Efficiency Considerations

### When $N$ Is Small

As we have stressed throughout this chapter, our analysis of efficiency has been based on the number of comparisons made by a sorting algorithm. This number gives us a rough estimate of the computation time involved. The other activities that accompany the comparison (swapping, keeping track of Boolean flags, and so forth) contribute to the "constant of proportionality" of the algorithm.

In comparing Big-O evaluations, we ignored constants and smaller-order terms because we wanted to know how the algorithm would perform for large values of  $N$ . In general, an  $O(N^2)$  sort requires few extra activities in addition to the comparisons, so its

constant of proportionality is fairly small. On the other hand, an  $O(N \log_2 N)$  sort may be more complex, with more overhead and thus a larger constant of proportionality. This situation may cause anomalies in the relative performances of the algorithms when the value of  $N$  is small. In this case  $N^2$  is not much greater than  $N \log_2 N$ , and the constants may dominate instead, causing an  $O(N^2)$  sort to run faster than an  $O(N \log_2 N)$  sort.

### *Eliminating Calls to Procedures and Functions*

#### **In-Line Subprograms**

You may have seen versions of some of the sorting procedures developed in this chapter written without local subprograms. Instead, the statements of the local subprograms are written right in the body of the sort procedure. For example, we could replace every occurrence of

```
Swap (Left => X, Right => Y);
```

with the in-line expansion:

```
Temp := X;
X := Y;
Y := Temp;
```

Similarly, in `Selection_Sort`, we coded the operation to find the minimum element as a nested function, `Min_Index`; and in `Bubble_Sort`, we coded a nested procedure `Bubble_Up`. We could expand all of these subprograms in-line right in the sort procedure bodies.

Of course coding these operations as procedures and functions makes the code simpler to write and to understand, avoiding more complicated nested loop structures. Why would a programmer give up the readability we get from procedural abstraction? The answer usually given is that by eliminating the extra overhead of subprogram calls, the program will run slightly faster. Although it may improve the running time, writing the code for these local procedures directly in-line does not effect the Big-O of the algorithm—only the constants of proportionality are changed.

Because we believe that a programmer's time is valuable, we prefer to write our programs with emphasis on clarity. This choice usually means that each different level module in our algorithm design is coded as a separate subprogram. However, with Ada this does not necessarily mean that our programs run slower than those written with fewer subprograms. Ada provides a mechanism called *pragmas* that we can use to direct the compiler to make certain changes to our program when it translates it into machine language. The Ada Language Reference Manual describes over a dozen predefined pragmas. We discuss one pragma that is relevant here: pragma `INLINE`.

Pragma `INLINE` directs the Ada compiler to substitute a copy of the subprogram body for every call of the subprograms it names. For example, the statement

```
pragma INLINE (Swap, Min_Index);
```

written just after these two subprograms in procedure `Selection_Sort` directs the Ada compiler to substitute the three assignment statements in procedure `Swap` for every call to

that procedure and the six statements in function `Min_Index` for every call to that function. These substitutions are made in the machine code produced by the compiler, *not to our Ada source code*. As a result of the elimination of procedure calls, `pragma INLINE` gives us efficiency of in-line code while maintaining the procedural abstraction so important for a person reading the code. Using `pragma INLINE` does *not* impose any restrictions on how we write a subprogram. We can use whatever names we desire for formal parameters and local variables.

Programmers using a source-level debugger must often pay a price for the increase in efficiency achieved through `pragma INLINE`. By copying the code in-line, the Ada source program no longer "matches" the machine language program created by the compiler. Your debugger may no longer permit you to *step* through the in-lined subprograms or set *breakpoints* within them. We recommend that you only include `pragma INLINE` after you have successfully completed the debugging and testing of a particular program unit.

There is another, smaller problem with this pragma. An Ada compiler is free to ignore `pragma INLINE` in some or all cases. Check the manual for your Ada compiler to see how it handles `pragma INLINE`. And while you have the manual open to the section on pragmas, read about some of the other pragmas available for improving the efficiency of your Ada code.

### Removal of Recursion

The recursive sorting procedures `Merge_Sort` and `Quick_Sort` have a similar situation: they require the extra overhead involved in executing the recursive calls. You may want to avoid this overhead by coding nonrecursive versions of these procedures. Because the recursion in these sort procedures is not *tail recursion*, removing recursion requires stacking (see Chapter 11).

If the recursive calls are less efficient, why would anyone ever decide to use a recursive version of a sort? The decision involves a choice between types of efficiency. Up until now, we have only been concerned with minimizing computer time. Computers are becoming faster and cheaper; however, it's not at all clear that computer *programmers* are following that trend. Therefore in some situations programmer time may be an important consideration in choosing a sort algorithm and its implementation. In this respect, the recursive version of `Quick_Sort` is more desirable than its nonrecursive counterpart, which requires the programmer to simulate the recursion explicitly.

### More about Sorting in General

We can instantiate procedures from the generic sort procedures developed in this chapter to solve a wide variety of sorting problems. In the following sections, we look at instantiation techniques for solving some of the more common sorting problems.

#### Sorting in Descending Order

All of our sorting procedures order the array elements in ascending order. Given a little time, we could rewrite any of them to order the elements in descending value. But instead of writing one generic procedure to sort into ascending order and another to sort into descending order, we can make use of the same "trick" we used with our generic priority queue: associate the ">" operator with the procedure's formal generic parameter "<". Here, for example, are the declarations for instantiating a descending order heap sort:

```

type Int_Array is array (Positive range <>) of Integer;

function Identity (Element : in Integer) return Integer;

procedure Descending_Sort is new Heap_Sort (Element_Type => Integer,
 Array_Type => Int_Array,
 "<" => ">");

```

### Keys

In our descriptions of the various sorts, we showed examples of sorting integers. In reality one is more likely to be sorting arrays of records that contain several fields of information, ordering the records according to a *key* field. Each record must contain some identifying key, such as an `ID_Number` field. In addition, a record may contain secondary keys, which may or may not be unique.

We use the following declarations to illustrate how our generic sort procedures may be used to sort on any combination of keys:

```

subtype Name_String is String (1..10);
subtype Address_String is String (1..30);
subtype City_String is String (1..15);
subtype State_String is String (1..2);
subtype Zip_String is String (1..5);

type Name_Rec is
 record
 First : Name_String;
 Last : Name_String;
 end record;

type Mailing_Label_Rec is
 record
 Name : Name_Rec;
 Address : Address_String;
 City : City_String;
 State : State_String;
 Zip : Zip_String;
 end record;

-- Types for a list of club members
type Label_Array is array (Positive range <>) of Mailing_Label_Rec;
type Club_List (Max_Size : Positive) is
 record
 Length : Natural := 0;
 Info : Label_Array (1..Max_Size);
 end record;

Geology_Club : Club_List (Max_Size => 100);

```

Now suppose we want to send out meeting notices to all of the members of the Geology Club. Because we get a discount if all of our mail is sorted by zip code, we would like

to sort this list in ascending order by zip code. The following declarations instantiate a procedure to do just that:

```
function "<" (Left : in Mailing_Label_Rec; Right : in Mailing_Label_Rec)
 return Boolean is
begin
 return Left.Zip < Right.Zip;
end "<";

procedure Sort_Zip is new Some_Sort (Element_Type => Mailing_Label_Rec,
 Array_Type => Label_Array,
 "<" => "<");
```

Now suppose we need an alphabetical listing of the names of the club members. This sort involves two different keys—the last name (primary key) and the first name (secondary key). To instantiate the desired sort, we need only write a "<" function that compares two elements using these keys. Here are the declarations:

```
function "<" (Left : in Mailing_Label_Rec; Right : in Mailing_Label_Rec)
 return Boolean is
begin
 if Left.Name.Last < Right.Name.Last then
 return True;
 elsif Left.Name.Last > Right.Name.Last then
 return False;
 else -- Same last name, check the first name
 return Left.Name.First < Right.Name.First;
 end if;
end "<";

procedure Sort_Name is new Some_Sort (Element_Type => Mailing_Label_Rec,
 Array_Type => Label_Array,
 "<" => "<");
```

This logic can be extended to instantiate sort procedures that sort on *any* number of keys.

### ***Sorting Pointers***

Sorting large records using some kind of exchange sort may require a lot of computer time just to move sections of memory from one place to another every time we make a swap. This move time can be reduced by setting up an array of pointers (either access values or array indices) to the records and then sorting the pointers instead of the actual records. This scheme is illustrated in Figure 14-20.

Note that after the sort the records are still in the same physical arrangement, but they may be accessed in order through the sorted array of pointers. Here are the declarations to instantiate a procedure to sort an array of access values designating mailing label records:

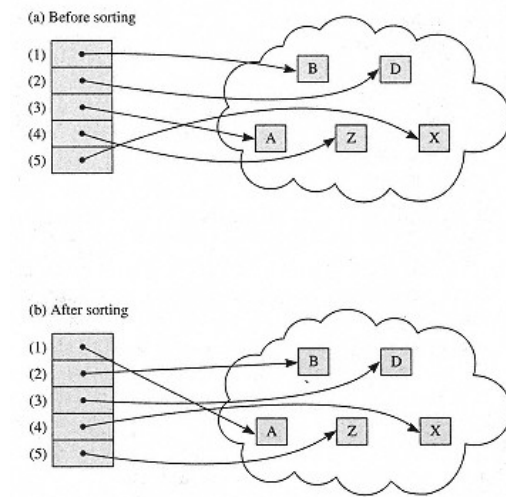


Figure 14-20  
Sorting Arrays with Access Values

```
type Label_Ptr is access Mailing_Label_Rec;
```

```
type Access_Array is array (Positive range <>) of Label_Ptr;
```

```
function "<" (Left : in Label_Ptr; Right : in Label_Ptr) return Boolean is
begin
 return Left.all.Zip < Right.all.Zip;
end "<";
```

```
procedure Sort_Zip is new Some_Sort (Element_Type => Label_Ptr,
 Array_Type => Access_Array,
 "<" => "<");
```

This scheme may be extended to allow us to keep a large array of data sorted on more than one key. For instance, with the declarations

```
type Employee_Rec is
 record
 ID_Num : Positive;
 Name : Name_String;
 Salary : Positive;
 end record;

subtype Index_Type is Positive range 1..5;
type Employee_Array is array (Positive range <>) of Employee_Rec;
type Pointer_Array is array (Positive range <>) of Index_Type;
```



```

Employee_Data : Employee_Array (Index_Type);
Name_Ord : Pointer_Array (Index_Type);
Sal_Ord : Pointer_Array (Index_Type);

```

the data in Figure 14-21 is physically stored according to the primary key, ID\_Num. The arrays Name\_Ord and Sal\_Ord contain pointers (indexes) to the records in the large array, Employee\_Data. In this way we can keep the array ordered with respect to the secondary keys, Name and Salary, as well.

Here are the declarations necessary to instantiate a procedure to sort an array of indices in order by the names they reference in the data array Employee\_Data:

| Employee_Data |                       | Name_Ord |   |
|---------------|-----------------------|----------|---|
| (1)           | 15<br>JONES<br>20000  | (1)      | 3 |
| (2)           | 20<br>SMITH<br>40000  | (2)      | 4 |
| (3)           | 34<br>ABLE<br>29246   | (3)      | 1 |
| (4)           | 100<br>BAKER<br>20000 | (4)      | 5 |
| (5)           | 144<br>OWENS<br>6000  | (5)      | 2 |

| Sal_Ord |   |
|---------|---|
| (1)     | 5 |
| (2)     | 1 |
| (3)     | 4 |
| (4)     | 3 |
| (5)     | 2 |

Figure 14-21  
Data Sorted on More Than One Key

```

function Smaller_Name (Left : in Index_Type; Right : in Index_Type)
 return Boolean is
 -- This function accesses the array Employee_Data globally.
begin
 return Employee_Data(Left).Name < Employee_Data(Right).Name;
end Smaller_Name;

```

```

-- Instantiate a procedure to sort the pointers in an array so they are
-- ordered by the names they reference in the data array Employee_Data.
procedure Name_Sort is new Some_Sort (Element_Type => Index_Type;
 Array_Type => Pointer_Array;
 "<" => Smaller_Name);

```

## Radix Sort

We have placed radix sort in this final section because, unlike all of the other sorting algorithms we discussed in this chapter, radix sort is not a *comparison* sort. Radix sort doesn't compare two keys in the array. Therefore, we can't analyze the amount of work done in terms of comparisons. In fact, the only thing that radix sort has in common with the other sorts is that it's given an unordered array and returns it ordered.

The idea behind radix sort is to divide the values into as many sublists as there are possible alternatives for each *position* in the key. For example, if the key is an integer number, each position is a digit and has 10 possibilities: 0..9. If the key is a string of letters and case is not important, then each position has 26 possibilities: 'a'..'z'. The number of possibilities is called the *radix*. After subdividing the items into radix sublists, we combine them back into one list and repeat the process. If we begin with the least *significant* position in the key, regroup the items in order, and repeat the process as many times as there are positions in the key, moving to next most significant position each time, then the array is ordered when we finish.

Let's illustrate the algorithm by sorting three-digit positive integers. We divide these integers into 10 sublists based on the least significant digit (the units position). Let's create an array of queues to hold the sublists:

```
-- Instantiate a queue package for integer elements
package Int_Queue is new Queue (Element_Type => Positive);

type Queue_Array is array (0..9) of Int_Queue.Queue_Type;

Queues : Queue_Array; -- An array of 10 queues
```

All items with a 0 in the units position are enqueued into `Queues (0)`; all items with a 1 in the units position are enqueued into `Queues (1)`; and so on. After the first pass through the array, we collect the sublists with those in `Queues (0)` first and those in `Queues (9)` last.

This algorithm is illustrated in Figures 14-22 and 14-23; only the keys are shown. Look at the keys after each pass; the digits in the column that correspond to the pass number are ordered (Figure 14-22). Likewise, the digits in the pass number position are the same as the index of the queue that it's in (Figure 14-23).

| Original<br>order | Order<br>after<br>1st pass | Order<br>after<br>2nd pass | Order<br>after<br>3rd pass |
|-------------------|----------------------------|----------------------------|----------------------------|
| 762               | 800                        | 800                        | 001                        |
| 124               | 100                        | 100                        | 100                        |
| 432               | 761                        | 001                        | 124                        |
| 761               | 001                        | 402                        | 402                        |
| 800               | 762                        | 124                        | 432                        |
| 402               | 432                        | 432                        | 761                        |
| 976               | 402                        | 761                        | 762                        |
| 100               | 124                        | 762                        | 800                        |
| 001               | 976                        | 976                        | 976                        |
| 999               | 999                        | 999                        | 999                        |

Figure 14-22  
Key Fields After Each Pass

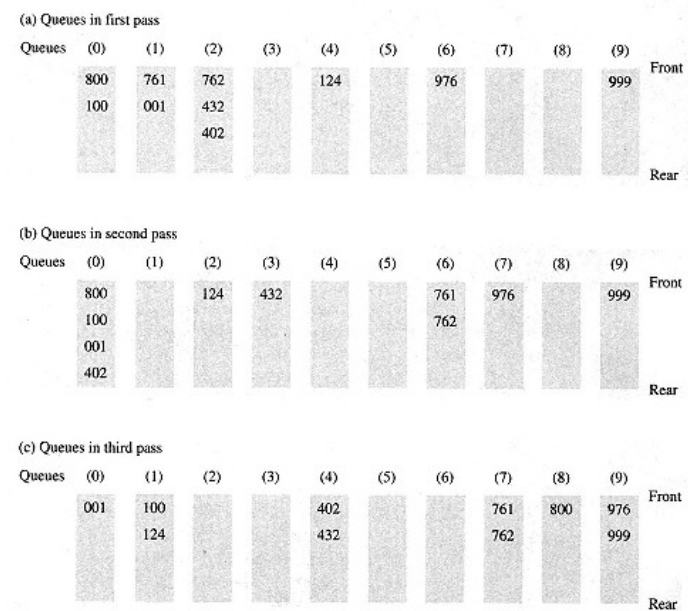


Figure 14-23  
Queues After Each Pass

Let's first write the algorithm for radix sort that matches our example:

### Radix Sort

```

for Position in 1..3 loop
 for Index in Info'Range
 Which_Queue := Digit at Position in the key of Info(Index)
 Enqueue (Queue => Queues(Which_Queue), Item => Info (Index))
 end loop
 Collect Queues
end loop

```

Each iteration of the outer loop in this algorithm corresponds to one pass in Figures 14-22 and 14-23. In the first pass we use the units digit of our integer key to determine in which queue to enqueue the element. In the second pass we use the tens digit and in the third pass we use the hundreds digit. Next we need to expand the Collect Queues step of the algorithm. This is where we collect the elements from all of the queues. We put them back into Info:

### Collect Queues

```

Info_Index := Info'First
for Queue_Index in 0..9 loop

```

```

loop
 exit when Queue_Empty (Queues(Queue_Index))
 Dequeue (Queue :=> Queues (Queue_Index), Item => Info (Info_Index))
 Info_Index := Info_Index + 1
end loop
end loop

```

The outer loop in this algorithm goes through our 10 queues. The inner loop takes all the elements from a queue and puts them back into the array Info.

### ***The Implementation Level***

Now that we understand the algorithm, let's look at how we can make it general. What generic formal parameters are necessary to implement a generic radix sort? As with all our sorts, we need a type for the elements we are sorting (*Element\_Type*) and a type for the array in which they are stored (*Array\_Type*).

For our radix sort algorithm, we must know the number of different positions in the key. We use a generic formal object called *Num\_Positions* to supply this information. Like we did in our example, let's number these positions starting with 1 (the least significant position) to *Num\_Positions* (the most significant position).

We divide the key into *Num\_Positions* subkeys. Because we work with subkeys of an element's key, we need a type for a subkey. This type must be supplied by the user; let's call it *Subkey\_Type*. Because we use the portions of a key as indices in our array of queues, *Subkey\_Type* must be a discrete type. We have a type for a *portion* of a key. Do we need a type for the *entire* key? No. Nowhere in our algorithm do we use the entire key.

In our other generic sorts, we had two generic formal functions. One of these, ("*<*"), is used to compare two key values. Radix sort does not compare key values so we don't need the user to supply such a function. The second generic formal function returns the key of a given element. Again, radix sort does not work with the entire key. However, it does work with portions of the key, values of type *Subkey\_Type*. So we need the user to supply a function that returns a subkey for a given element. Because each element has *Num\_Positions* subkeys, we need to also supply the position of the subkey we want as a parameter. Specification 14-2 is our declaration for a generic radix sort.

### **Specification 14-2— Radix Sort**

generic

```

Num_Positions : in Positive; -- Number of positions in each element's key
type Element_Type is private;
type Array_Type is array (Positive range <>) of Element_Type;
type Subkey_Type is (<>);
with function Subkey_Of (Element : in Element_Type;
 Position : in Positive) return Subkey_Type
procedure Radix_Sort (Info : in out Array_Type);
-- Sorts an array of elements into ascending order

-- Preconditions: Lowest value of Position is least significant subkey and
-- Highest value of Position is most significant subkey
-- Postconditions: Elements in Info are in ascending order by key

```

And here are the declarations necessary to instantiate a radix sort for our example (sorting an array of three-digit non-negative integers):

```
subtype Element_Type is Integer range 0..999;
type Element_Array is array (Positive range <>) of Element_Type;
subtype Digit_Type is Integer range 0..9;
subtype Digit_Position is Positive range 1..3;

function Digit_Of (Number : in Element_Type;
 Position : in Positive) return Digit_Type;

procedure Sort is new Radix_Sort (Num_Positions => 3,
 Element_Type => Element_Type,
 Array_Type => Element_Array,
 Subkey_Type => Digit_Type,
 SubKey_Of => Digit_Of);
```

Before we go on to the body of the radix sort procedure, let's implement the body of function `Digit_Of`. This function is given a three-digit number and a position. It returns the digit at the given position. This task was simple to do by hand in our example. How do we do it in Ada? We use the `/` and `rem` operators. Let's calculate a few positions and look for a pattern. Assume the `Number` is 749.

Digit at position 1:       $(\text{Number} / 1) \text{ rem } 10 = 9$

Digit at position 2:       $(\text{Number} / 10) \text{ rem } 10 = 4$

Digit at position 3:       $(\text{Number} / 100) \text{ rem } 10 = 7$

Here is the resulting function:

```
function Digit_Of (Number : in Element_Type;
 Position : in Positive) return Digit_Type is
begin
 return (Number / 10 ** (Position - 1)) rem 10;
end Digit_Of;
```

Body 14-8 implements our radix sort algorithm.

#### Body 14-8—

#### Radix Sort Implements Specification 14-2

```
with Queue;
procedure Radix_Sort (Info : in out Array_Type) is

 -- Instantiate a queue package for Elements
 package Radix_Queue is new Queue (Element_Type => Element_Type);

 -- An array of queues for the sublists
 type Queue_Array is array (Subkey_Type) of Radix_Queue.Queue_Type;
 Queues : Queue_Array;
```

```

Which_Queue : Subkey_Type;

procedure Collect_Queues (Info : out Array_Type;
 Queues : in out Queue_Array) is
-- Transfers all the elements from the Queues into the array

-- Preconditions: The total number of elements in Queues = Info'Length

-- Postconditions: Info contains the elements in Queues with the elements taken
-- in order from Queues(Queues'First) to Queues(Queues'Last).
-- All queues are empty.
 Info_Index : Positive; -- Index for array Info
begin
 Info_Index := Info'First;
 -- Each iteration, one queue is emptied, its elements moved to Info
 for Queue_Index in Queues'Range loop
 loop -- Each iteration, one element is dequeued into Info
 exit when Radix_Queue.Empty (Queues(Queue_Index));
 Radix_Queue.Dequeue (Queue => Queues(Queue_Index), Item => Info(Info_Index));
 Info_Index := Info_Index + 1;
 end loop;
 end loop;
end Collect_Queues;

begin -- Radix Sort
 -- Each iteration, all elements are sorted by one subkey
 for Position in 1..Num_Positions loop
 -- Each iteration, one element is copied into the appropriate subkey queue
 for Index in Info'Range loop
 -- Determine which queue to place the Index'th element
 Which_Queue := Subkey_Of (Element => Info(Index), Position => Position);
 -- Enqueue the element
 Radix_Queue.Enqueue (Queue => Queues(Which_Queue), Item => Info(Index));
 end loop;
 -- Combine the information in all the queues
 Collect_Queues (Info, Queues);
 end loop;
end Radix_Sort;

```

### ***Analysis of Radix Sort***

We said earlier that we can't use the number of comparisons as a basis for analyzing the amount of work done in radix sort because there aren't any comparisons. We can, however, analyze the number of times each item is processed and use this measure for the amount of work done. Each item in `Info` is processed `Num_Positions` times. If we use  $N$  for the number of elements and  $K$  for the number of positions, this processing is  $O(KN)$ . The

processing includes extracting a subkey from the element, inserting the element into a queue, and dequeuing each element. Each of these processing steps is  $O(1)$ , so the Big-O of radix sort is  $O(KN)$ . Although we cannot directly compare radix sort with the other sorts, we can say that the "shorter" the key, the better radix sort performs in comparison with the other sorts.

What about space requirements? Our `Radix_Sort` procedure requires space for at least two copies of each element: one place in the array and one space in a queue. If the queues are array-based, the amount of space is prohibitive because there must be room in *each* queue for *every* element. If the queues are linked, additional space for  $N$  pointers is required.

### Summary

We have not attempted in this chapter to give every known sorting algorithm. We have presented a few of the popular sorts, of which many variations exist. It should be clear from this discussion that no single sort is best for all applications. The simpler, generally  $O(N^2)$  sorts work as well, and sometimes better, for fairly small values of  $N$ . Because they are simple, these sorts require relatively little programmer time to write and maintain. As you add features to improve sorts, you also add to the complexity of the algorithms, increasing both the work required by the routines and the programmer time needed to maintain them.

Another consideration in choosing a sort algorithm is the order of the original data. If the data are already ordered (or almost ordered), `Short_Bubble_Sort` is  $O(N)$ , whereas some versions of a `Quick_Sort` are  $O(N^2)$ .

As always, the first step in choosing an algorithm is to determine the goals of the particular application. This step usually narrows the options considerably. After that, knowledge of the strong and weak points of the various algorithms assists you in making a choice.

The following table summarizes the comparison among the sorts discussed in this chapter in terms of Big-O.

**Big-O Comparison of Sorting Algorithms**

| Sort                            | Order of Magnitude |                 |                 |
|---------------------------------|--------------------|-----------------|-----------------|
|                                 | Best Case          | Average         | Worst Case      |
| <i>Simple Sorts</i>             |                    |                 |                 |
| <code>Selection_Sort</code>     | $O(N^2)$           | $O(N^2)$        | $O(N^2)$        |
| <code>Bubble_Sort</code>        | $O(N^2)$           | $O(N^2)$        | $O(N^2)$        |
| <code>Short_Bubble_Sort</code>  | $O(N)$             | $O(N^2)$        | $O(N^2)$        |
| <code>Insertion_Sort</code>     | $O(N)$             | $O(N^2)$        | $O(N^2)$        |
| <i>Divide-and-Conquer Sorts</i> |                    |                 |                 |
| <code>Merge_Sort</code>         | $O(N \log_2 N)$    | $O(N \log_2 N)$ | $O(N \log_2 N)$ |
| <code>Quick_Sort</code>         | $O(N \log_2 N)$    | $O(N \log_2 N)$ | $O(N^2)$        |
| <code>Heap_Sort</code>          | $O(N \log_2 N)$    | $O(N \log_2 N)$ | $O(N \log_2 N)$ |

So that you can see that there really is a difference between an  $O(N^2)$  sort and an  $O(N \log_2 N)$  sort, we ran some timed tests. With a list of 1000 elements, we executed each of the sorts in this chapter, first on an array of mixed-up records, then on a sorted array of records, and then on array of records in reverse order. The records sorted contained a 5- digit positive integer key and a 20-character string. `Pragma INLINE` was *not* used in these tests. The results of the tests are summarized in the next table. Perhaps the only unexpected results are those from `Short_Bubble_Sort`. The time to sort a previously sorted list is very low, but the additional if statement and two assignment statements increased the times (on this computer) for random and reverse lists significantly when compared to the original bubble sort.

**Timed Comparison of Sorting Procedures (CPU time in seconds)**

| <i>Sort</i>                                          | <i>Mixed-up</i> | <i>In-Order</i> | <i>Reverse Order</i> |
|------------------------------------------------------|-----------------|-----------------|----------------------|
| <i>Simple Sorts</i>                                  |                 |                 |                      |
| <code>Selection_Sort</code>                          | 1.11            | 1.12            | 1.11                 |
| <code>Bubble_Sort</code>                             | 1.69            | 0.63            | 2.60                 |
| <code>Short_Bubble_Sort</code>                       | 2.72            | 0.01            | 4.15                 |
| <code>Insertion_Sort</code>                          | 0.56            | 0.01            | 1.09                 |
| <i>Divide and Conquer Sorts</i>                      |                 |                 |                      |
| <code>Merge_Sort</code>                              | 0.10            | 0.08            | 0.09                 |
| <code>Quick_Sort</code> (first element split value)  | 0.06            | 1.27            | 1.54                 |
| <code>Quick_Sort</code> (middle element split value) | 0.04            | 0.03            | 0.03                 |
| <code>Heap_Sort</code>                               | 0.23            | 0.24            | 0.22                 |
| <code>Radix_Sort</code>                              | 0.13            | 0.08            | 0.13                 |

These Big-O comparisons and timing experiments can help us choose the best sort for our problem. The simple sorts are easy to write and useful when  $N$  is small. Of these, `Insertion_Sort` provides the best performance and `Bubble_Sort` the worst. `Quick_Sort` lives up to its name as the quickest of the divide-and-conquer sorts. However, because it is sensitive to the initial order of the data, there is always a possibility that for some data its timing may be closer to  $O(N^2)$ . If your application can't tolerate this possibility, use the `Merge_Sort` or `Heap_Sort`. `Merge_Sort` may be a little faster than `Heap_Sort`, but it requires twice as much memory.

## Exercises

1. Show the contents of the array

|     |     |     |     |     |     |     |     |     |      |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|
| 43  | 7   | 10  | 23  | 18  | 4   | 19  | 5   | 66  | 14   |
| (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) | (9) | (10) |



after the fourth iteration of

- (a) Bubble\_Sort
- (b) Selection\_Sort

2. (a) Show how the values in the array in Exercise 1 would have to be rearranged to satisfy the heap property.

(b) Show how the heap would look with four values in place, after reheaping.

3. Show how the values in the array in Exercise 1 would be arranged immediately before the execution of procedure Merge in the original (nonrecursive) call to Merge\_Sort.

4. You are given this array:

|     |     |     |     |     |     |     |     |     |      |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|
| 26  | 24  | 3   | 17  | 25  | 24  | 13  | 60  | 47  | 1    |
| (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) | (9) | (10) |

Tell which sorting algorithm would produce the following results after four iterations:

(a)

|     |     |     |     |     |     |     |     |     |      |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|
| 1   | 3   | 13  | 17  | 26  | 24  | 24  | 25  | 47  | 60   |
| (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) | (9) | (10) |

(b)

|     |     |     |     |     |     |     |     |     |      |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|------|
| 1   | 3   | 13  | 17  | 25  | 24  | 24  | 60  | 47  | 26   |
| (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) | (9) | (10) |

5. How many comparisons would be needed to sort an array containing 100 elements using Short\_Bubble\_Sort

- (a) in the worst case?
- (b) in the best case?

6. A sorting procedure is called to sort a list of 100 integers that have been read from a file. If all 100 values are zero, what would the execution requirements (in terms of Big-O) be if the sort used was each of the following?

- (a) Quick\_Sort, with the first element used as the split value
- (b) Short\_Bubble\_Sort
- (c) Selection\_Sort
- (d) Heap\_Sort
- (e) Merge\_Sort

7. How many comparisons would be needed to sort an array containing 100 elements using Selection\_Sort if the original array values were already sorted?

- (a) 10,000
- (b) 9900
- (c) 4950
- (d) 99
- (e) None of the above

8. A merge sort is used to sort an array of 1000 test scores in descending order. Which of the following statements is true?
- (a) The sort is fastest if the original test scores are ordered from smallest to largest.
  - (b) The sort is fastest if the original test scores are in completely random order.
  - (c) The sort is fastest if the original test scores are ordered from largest to smallest.
  - (d) The sort is the same, no matter what the order of the original elements.
9. A list is ordered from smallest to largest when a sort is called. Which of the following sorts would take the longest time to execute and which would take the shortest time?
- (a) Heap\_Sort
  - (b) Short\_Bubble\_Sort
  - (c) Quick\_Sort (with the first element as the split value)
  - (d) Selection\_Sort
10. (a) In what case(s), if any, is the bubble sort  $O(N)$ ?
- (b) In what case(s), if any, is the selection sort  $O(\log_2 N)$ ?
  - (c) In what case(s), if any, is quick sort  $O(N^2)$ ?
11. A very large array of elements is to be sorted. The program is to be run on a personal computer with limited memory. Which sort would be a better choice: heap sort or merge sort? Why?
12. Use the Three-Question Method to verify the Merge\_Sort procedure.
13. *True or False?* If you answer false, correct the statement.
- (a) Merge sort requires more space to execute than heap sort.
  - (b) Quick sort (using the first element as the split value) is better for nearly sorted data than is heap sort.
  - (c) The efficiency of heap sort is not affected by the order of the elements on entrance to the procedure.
14. Which is true about quick sort?
- (a) A recursive version executes faster than a nonrecursive version.
  - (b) A recursive version has fewer lines of code than a nonrecursive version.
  - (c) A nonrecursive version takes more space on the run-time stack than a recursive version.
  - (d) It can only be programmed as a recursive procedure.
15. What is meant by the statement that programmer time is an efficiency consideration? Give an example of a situation in which programmer time is used to justify the choice of an algorithm, possibly at the expense of other efficiency considerations.
16. Identify one or more correct answers: Sorting an array of pointers to list elements, rather than sorting the elements themselves, is a good idea when
- (a) the number of elements is very large.
  - (b) the individual elements are large in size.
  - (c) the sort is recursive.
  - (d) there are multiple keys on which to sort the elements.
17. Study the Ada code for procedure Quick\_Sort.
- (a) In procedure Split, why was Left initialized to Info'First + 1 rather than to Info'First?
  - (b) Why was or else used rather than or in the exit statement of the first inner loop in procedure Split?
18. (a) Give arguments for and against using procedures (such as Swap) to encapsulate frequently used code in a sorting procedure.
- (b) How does the availability of pragma INLINE modify these arguments?

19. Write the necessary declarations to instantiate a version of the bubble sort algorithm that sorts a list of integers in descending order.
20. The insertion sort developed in this chapter uses a sequential search to find the insertion location in the sorted part of the array.
  - (a) Rewrite procedure `Insertion_Sort` using a binary search instead of the sequential search.
  - (b) In terms of comparisons, what is the Big-O for this version of `Insertion_Sort`?
  - (c) In terms of data movements, is there any difference between this version of `Insertion_Sort` and the one presented in the chapter?
21. In the shaker sort version of the bubble sort, the smallest element bubbles to the top during the first iteration, but during the second iteration the largest element bubbles to the bottom. The sort alternates bubbling up and bubbling down until the list is sorted.
  - (a) Modify `Bubble_Sort` to create `Shaker_Sort`.
  - (b) Describe this algorithm in terms of Big-O.
22. (a) Assuming your Ada compiler does not support `pragma INLINE`, write a version of `Selection_Sort` that does not make any procedure calls.
  - (b) Name an advantage and disadvantage to using this version of `Selection_Sort`.
23. Write a recursive version of `Selection_Sort`.
24. Write a procedure `Sort_Linked` that sorts the elements in an unordered linked list. (Use the declarations of a linear singly linked list from Chapter 8.)
25. Quick sort works best for large randomly ordered lists, but when the sublists become small, it might be more efficient to call another sort. Modify `Quick_Sort` to make the recursive calls only when the number of elements to be sorted on that call is greater than a constant called `Limit`; otherwise, it calls `Insertion_Sort`.
26. Modify the `Split` procedure in `Quick_Sort` so that the split value is the middle value in the array rather than the first value.

Use the following declarations for Exercises 27–29:

```

type Suit_Type is (Club, Diamond, Heart, Spade);
type Number_Type is (Two, Three, Four, Five, Six, Seven, Eight,
 Nine, Ten, Jack, Queen, King, Ace);
type Card_Type is
 record
 Suit : Suit_Type;
 Number : Number_Type;
 end record;

type Card_Array is array (Positive range <>) of Card_Type;
subtype Deck_Array is Card_Array (1..52);

```

27. Write the necessary declarations to instantiate a heap sort procedure that can be used to sort a deck of cards in order by card numbers. Ignore the cards' suit.
28. Write a "<" function that is given two cards of `Card_Type` and returns True if the first is less than the second. Use the card suit as the primary key and the card number as the secondary key for this comparison.
29. Write the necessary declarations to instantiate a heap sort procedure that can be used to sort a deck of cards in order by suit (primary key) and number (secondary key). You can use the function you wrote for the previous question.

Use the following declarations for Exercises 30–32:

```

Max_Students : constant := 1000;

subtype String_Type is String (1..20);
type Class_Type is (Freshman, Sophomore, Junior, Senior);
subtype GPA_Type is Float range 0.0 .. 4.0;

type Name_Type is
 record
 First_Name : String_Type;
 Last_Name : String_Type;
 end record;

type Student_Type is
 record
 Student_Name : Name_Type;
 Student_ID : Positive;
 GPA : GPA_Type;
 Hours_Earned : Natural;
 Class : Class_Type;
 Zip_Code : Positive;
 end record

type Student_Array is array (Positive range <>) of Student_Type;

type Student_List (Max_Size : Positive) is
 record
 Num_Students : Natural := 0;
 Students : Student_Array (1..Max_Size);
 end record;

Students : Student_List (Max_Size => Max_Students)

```

30. The list of student records is sorted according to the `Student_ID` field as the primary key. For some types of processing, we need to order the student data according to a secondary key, `Student_Name`. To save space, we decide to keep an array of index pointers ordered on the name field.

(a) Write the necessary declarations for a *list* of index pointers.

(b) Write the code to initialize the list of index pointers so the first value is 1, the second is 2, ..., the last is `Students.Num_Students`.

31. (a) Write the necessary declarations (including any comparison functions) necessary to instantiate a quick sort procedure to sort the array in the list of pointers you declared for the previous exercise so that they designate the student records in ascending order.

(b) Write a sample call of the procedure you instantiated that sorts the list of index pointers you declared in Exercise 30.

32. Write a procedure to display the names and GPA of all the elements in the list `Students` sorted by GPA. The parameters to this procedure should be `Students` and the list of index pointers you sorted in Exercise 30. Neither parameter should be changed by the procedure.

33. Sooe County is about to have its annual Big Pig Contest. Because the sheriff's son, Wilbur, is majoring in computer science, the county hires him to computerize the Big Pig judging. Each pig's name (string) and weight (integer) is to be read in from the keyboard. The county expects 500 entries this year.

The output needed is a listing of the 10 heaviest pigs, ordered from biggest to smallest. Because Wilbur has just learned some sorting methods in school, he feels up to the task of writing this "pork-gram." He writes a program to read in all the entries into an array of records, then uses a selection sort to put the whole array in order according to the `Pig_Weight` field. He then prints the 10 largest values from the array.

Can you think of a more efficient way to write this program? If so, write the algorithm.

34. State University needs a listing of the overall SAT percentiles of the 14,226 students it has accepted in the past year. The data is in a text file, with one line per student. That line contains the student's ID number, SAT overall percentile, math score, English score, and high school grade point average. (There is at least one blank between each two fields.) The output needed is a listing of all the percentile scores, one per line, ordered from highest to lowest. Duplicates should be printed. Write an  $O(N)$  procedure to produce the listing.

35. Why does the heap sort have the precondition that `Info'First` is equal to 1?

36. Determine the Big-O of `Selection_Sort`, `Bubble_Sort`, `Insertion_Sort`, `Merge_Sort`, `Quick_Sort`, and `Heap_Sort` based on the number of elements moved rather than on the number of comparisons.

(a) For the best case.

(b) For the worst case.

37. Write the necessary declarations (and function body) to instantiate a radix sort for an array of part names. Each part name has 10 characters. This sort should consider equivalent upper- and lowercase letters to be the same.

38. What simple change can be made to the radix sort algorithm developed in this chapter to convert it from one that sorts in ascending order to one that sorts in descending order?

39. The following procedure sorts an array of integer values into ascending order.

```
type Int_Array is array (Positive range <>) of Integer;

procedure Integer_Sort is (Info : in out Int_Array) is
 Result : Int_Array (Info'Range);
 Result_Index : Natural;
begin
 Result_Index := Result'First - 1;
 for Value in Integer'First .. Integer'Last loop
 for Info_Index in Info'Range loop
 if Info(Info_Index) = Value then
 Result_Index := Result_Index + 1;
 Result(Result_Index) := Info(Info_Index);
 end if;
 end loop;
 end loop;
 Info := Result;
end Integer_Sort;
```

(a) Explain why `Integer_Sort` is an  $O(N)$  sort.

(b) Would this  $O(N)$  procedure sort an array of 10 integers faster than the  $O(N^2)$  sorts discussed in this chapter?

## Programming Problems

1. Implement and test an insertion sort procedure that sorts a linked list of values. Optional: Make the procedure a generic procedure.
2. Implement and test a quick sort procedure that sorts a doubly linked list of values. Optional: Make the procedure a generic procedure.
3. The object of this programming assignment is twofold. First, you are to compare the relative performance of different sorting algorithms on the same data sets. Second, you are to compare the relative performance of the same algorithm on different size data sets.

You are to test five sorting algorithms:

1. Selection sort
2. Bubble sort (original version)
3. Merge sort
4. Quick sort (middle element split version)
5. Heap sort

You must modify each sort to include two counter parameters. One counter returns the number of comparisons made and the second counter returns the number of element movements.

### Input

Two files containing integers to be sorted:

1. a file of 1000 integers in random order
2. a file of 1000 integers already in order

If your instructor does not provide these files, you can write a program to produce them. You can use the random number generator given in Appendix K to create files of integers in random order.

### Processing

For each sort algorithm, read into an array and sort the first 10 numbers in each file. Repeat this for the first 50, 100, 500, and then all 1000 numbers in each file.

### Output

A neatly formatted table containing the following information:

1. the name of the sort
2. the order of data (random or in order)
3. the number of elements sorted
4. the number of comparisons required to sort the elements
5. the number of elements moved to sort them

### Turn In

1. program listing
2. program output listed above
3. a graph of the number of comparisons (vertical axis) versus the number of elements sorted (Plot the data for all five sorts on a single graph.)

4. a graph of number of data movement (vertical axis) versus the number of elements sorted (Plot the data for all five sorts on a single graph.)

***Notes***

This problem may be divided up among class members. Each person in the class can be assigned one sort algorithm.

## 15— Searching

### GOALS

After reading this chapter, you should be able to

- discuss the performances of the following search algorithms:  
sequential search of an unordered list  
sequential search of an ordered list  
binary search
- design and implement a high-probability ordered list
- define the following terms:  
hashing  
rehashing  
collisions  
linear probing  
clustering
- design and implement an appropriate hashing function for an application
- design and implement a collisions-resolution algorithm for a hash table
- discuss the efficiency considerations for the searching and hashing algorithms, in terms of Big-O



It has been said that a typical commercial program spends 40% to 60% of its time on searching. Because searching is so important we devote this entire chapter to it. We begin by reviewing the searching strategies developed earlier in the book and then present some new techniques.

As we discussed in Chapter 3, for each particular structure used to hold data, the functions that allow access to elements in the structure must be defined. In some cases access is limited to the elements in specific positions in the structure, such as the top element in a stack or the front element in a queue. Often, when data is stored in a list or a table, we want to be able to access any element in the structure.

Sometimes the retrieval of a specified element can be performed directly. For instance, the fifth element of the list stored *sequentially* in an array-based list called `List` is found in `List.Info(5)`. Often, however, you want to access an element according to some key value. For instance, if a list contains student records, you may want to find the record of the student named Suzy Brown (the key field is `Student_Name`) or the record of the student whose ID number is 203557 (the key field is `ID_Num`). In cases like these, some kind of *searching technique* is needed to allow retrieval of the desired record. We review several algorithms for searching a list for an element with a specified key value. The list is ordered sequentially in an array, as described in Chapter 8. The declarations of this list type are repeated here:

```
type Element_Array is array (Positive range <>) of Element_Type;
```

```
type List_Type (Max_Size : Positive) is
 record
 Length : Natural := 0;
 Info : Element_Array (1..Max_Size);
 end record;
```

For each of the search techniques we discuss, we write a procedure that is given a slice of the list's storage array and a key to search for, and returns the location of the element with the desired key and a Boolean flag `Found`, indicating whether the key was in the list. Because the search procedures return an index into the array, they are *not* seen by the user of the List ADT. They are designed as auxiliary procedures to be used by any list operation that requires a search.

### Linear Searching

We cannot discuss efficient ways to find an element in a list without considering how the elements were inserted into the list. Therefore, our discussion of search algorithms is related to the issue of the List's Insert operation. Suppose that we want to insert elements as quickly as possible, and we are not as concerned about how long it takes to find them. We could use the following insertion algorithm:

#### Insert

```
Increment List.Length
Put element in List.Info (List.Length)
```

This is an  $O(1)$  insertion algorithm. The resulting list is ordered according to the time of insertion, not according to key value.

To search this list for the element with a given key, we must use a simple *sequential search*. Beginning with the first element in the list, we search for the desired element by examining each subsequent item's key until either the search is successful or the list is exhausted:

### Linear Search (unordered data)

```

Index := Info'First
loop
 exit when Index > Info'Last or else Key = Key_Of (Info(Index))
 Index := Index + 1
end loop
Found := Index <= Info'Last

```

We can code this algorithm as the following procedure:

```

procedure Linear_Search (Info : in Element_Array;
 Key : in Key_Type;
 Found : out Boolean;
 Location : out Positive) is
-- Searches for the location of Key in the array Info

-- Preconditions : None

-- Postconditions : If Key is in Info
-- Found is True
-- Location is the index of the element with Key
-- else
-- Found is False
-- Location is undefined

 Index : Positive;
begin
 Index := Info'First;
 loop -- Each iteration, one array element is checked
 exit when Index > Info'Last or else -- End of list?
 Key = Key_Of (Info(Index)); -- Found Key?
 Index := Index + 1;
 end loop;
 -- Assign results to out parameters
 Found := Index <= Info'Last;
 Location := Index;
end Search_Array;

```

Based on the number of comparisons, it should be obvious that this search is  $O(N)$ , where  $N$  represents the number of elements. In the worst case, in which we are looking for the last element in the list or for a nonexistent element, we have to make  $N$  key comparisons. On the average, assuming that there is an equal probability of searching for any item in the list, we make  $N/2$  comparisons for a successful search; that is, on the average we have to search half of the list.

### High-Probability Ordering

The assumption of equal probability for every element in the list is not always valid. Sometimes certain list elements are in much greater demand than others. This observation suggests a way to improve the search. We can put the most-often-desired elements at the beginning of the list. Using this scheme, you are more likely to make a hit in the first few tries, and rarely do you have to search the whole table.

If the elements in the list are not static or if you can't predict their relative demand, you need some scheme to keep the most frequently used elements at the front of the list. One way to accomplish this goal is to move each element accessed to the front of the list. Of course, there is no guarantee that this element is later frequently used. If the element is not retrieved again, however, it drifts toward the end of the list as other elements are moved to the front. This scheme is easy to implement for linked lists, requiring only a couple of pointer changes, but it is less desirable for lists kept sequentially in arrays because of the need to move all the other elements down to make room at the front.

A second approach, which causes elements to move toward the front of the list gradually, is appropriate for either linked or sequential list representations. As each element is accessed, it is swapped with the element that precedes it. Over many list retrievals, the most frequently desired elements tend to be grouped at the front of the list. To implement this approach, we only need to modify the end of the `Linear_Search` procedure to include

```
if Index <= Info'Last and then Key = Key_Of (Info(Index)) then
 Found := True;
 -- If the element isn't already at the top, move it up in the list
 if Index > Info'First then
 Swap (Info(Index), Info(Index - 1));
 Location := Index - 1;
 else
 Location := Index;
 end if;
else
 Found := False;
 Location := Index;
end if;
```

In the original version of `Linear_Search`, we passed `Info` as an `in` mode parameter. In this version, however, `Info` must be passed as an `in out` mode parameter, as it is actually changed by the search operation. This change should be documented; it is an unexpected side effect of searching the list.

Keeping the most active elements at the front of the list doesn't affect the worst case; if the search value is the last element or is not in the list, the search still takes  $N$  comparisons. This is still an  $O(N)$  search. The *average* performance on successful searches should be better, however. Both of these algorithms depend on the assumption that some elements in the list are used much more often than others. If this assumption isn't applicable, a different ordering strategy is needed to improve the efficiency of the search technique.

### Key Ordering

If a list is ordered according to the key value, we can write more efficient search routines. To support a key-ordered list, we must either insert the elements in order, as the `Insert`

procedure does in Chapter 8, or we must sort the list before searching it, as we discussed in Chapter 14. Note that inserting the elements in order is an  $O(N^2)$  process; as each insertion is  $O(N)$ . If we insert each element in the next free slot, and then sort the list with a "good" sort, the process is  $O(N \log_2 N)$ .

If the list is sorted, a sequential search no longer needs to search the whole list to discover that an element does not exist. It only needs to search until it has passed the element's logical place in the list—that is, until an element with a larger key value is encountered. The sequential and linked versions of the `Search_Array` and `Search_Linked_List` procedures in Chapter 8 implement this search technique. A version is shown below:

```
procedure Linear_Search2 (Info : in Element_Array;
 Key : in Key_Type;
 Found : out Boolean;
 Location : out Positive) is
-- Searches for the location of Key in the array Info

-- Preconditions : Info is sorted in ascending order

-- Postconditions : If Key is in Info
-- Found is True
-- Location is the index of the element with Key
-- else
-- Found is False
-- Location is the index of where Key
-- should be inserted

 Index : Positive;
begin
 Index := Info'First;
 loop -- Each iteration, one array element is checked
 exit when Index > Info'Last or else
 Key < Key_Of (Info(Index)) or else
 Key = Key_Of (Info(Index));
 Index := Index + 1;
 end loop;
 -- Assign results to out parameters
 Found := Index <= Info'Last and then
 Key = Key_Of (Info(Index));
 Location := Index;
end Linear_Search2;
```

The advantage of sequentially searching an ordered list is the ability to stop searching before the list is exhausted if the element does not exist. Again, the search is  $O(N)$ —the worst case, searching for the largest element, still requires  $N$  comparisons. The average number of comparisons for an unsuccessful search is now  $N/2$ , however, instead of a guaranteed  $N$ .

The advantage of sequential searching is its simplicity. The disadvantage is its performance: In the worst case you have to make  $N$  comparisons. If the list is sorted and stored in an array, however, you can improve the search time to a worst case of  $O(\log_2 N)$  by using a binary search. However, efficiency is improved at the expense of simplicity.

## Binary Searching

We know of a way to improve searching from  $O(N)$  to  $O(\log_2 N)$ . If the data elements are ordered and stored sequentially in an array, we can use a *binary* search. The binary search algorithm improves the search efficiency by limiting the search to the area where the element might be. The binary search algorithm takes a divide-and-conquer approach. It continually pares down the area to be searched until either the element is found or the search area is gone (the element is not in the list). We developed the `Binary_Search` procedure in Chapter 2, and converted it to a recursive procedure in Chapter 11. Here's the basic algorithm:

### Binary\_Search

```

if the list is empty
 stop searching -- value not in list
else
 Examine the middle element in the array
 if the middle element contains the desired key then
 stop searching -- value in list
 elsif the middle element is larger than the desired key then
 Search the first half of the array
 else (middle element is smaller than the key)
 Search the second half of the array
 end if
end if

```

The binary search, however, is not guaranteed to be faster for searching very small lists. Notice that even though the binary search generally requires fewer comparisons, each comparison involves more computation. When  $N$  is very small, this extra work (the constants and smaller terms that we ignore in determining the Big-O approximation) may dominate. Although fewer comparisons are required, each involves more processing. For instance, in one assembly language program, the linear search required 5 time units per comparison, whereas the binary search took 35. For a list size of 16 elements, therefore, the worst-case linear search would require  $5 * 16 = 80$  time units. The worst-case binary search only requires four comparisons, but at 35 time units each, the comparisons take 140 time units. In cases where the number of elements in the list is small, a linear search is certainly adequate and sometimes faster than a binary search.

As the number of elements increases, however, the disparity between the linear search and the binary search grows very quickly. Look back at Figure 2-7 to compare the rates of growth for the two algorithms.

How is this disparity reflected in actual executions of the linear and binary search procedures? The table below shows the results of timed tests that compared the performance of `Linear_Search2` and the `Binary_Search` (iterative version) from Chapter 2. Various sizes of arrays (as listed in the first column of the table) were loaded with integer values, and the search procedures were called to search for every value in the list. For instance, in the first test, an array of 1000 slots was loaded with the values 1 to 1000. The sequential search procedure was invoked repeatedly to search for list elements:

```

for Value in 1..List.Length loop
 Linear_Search2 (Info => List.Info (1..List.Length),
 Key => Value,
 Found => Found,
 Location => Location);
end loop

```

The elapsed time was calculated, and then the process was repeated for the binary search. The results columns show two sets of values for each search procedure: the total elapsed time for the set of searches and the average time per call. As you can see in the table, the average time to process each call to `Linear_Search2` doubles as the number of elements doubles; this result is what we would expect to see from an  $O(N)$  search. The average time to complete each call to the `Binary_Search` procedure rises much more slowly in comparison, consistent with what we would expect from an  $O(\log_2 N)$  search.

**Comparing the Linear and Binary Search Procedures**

| Number of<br>Elements in<br>List | Sequential Search  |                  | Binary Search      |                  |
|----------------------------------|--------------------|------------------|--------------------|------------------|
|                                  | Elapsed<br>Seconds | msec per<br>Call | Elapsed<br>Seconds | msec per<br>Call |
| 1000                             | 2.30               | 2.300            | 0.11               | 0.110            |
| 2000                             | 9.17               | 4.585            | 0.11               | 0.055            |
| 4000                             | 36.75              | 9.188            | 0.27               | 0.068            |
| 8000                             | 146.81             | 18.351           | 0.66               | 0.082            |

Note that the binary search discussed here is appropriate only for list elements stored in a sequential array-based representation. After all, how can you efficiently find the midpoint of a linked list? However, you already know of a structure that allows you to perform a binary search on a linked data representation, the binary search tree. The operations used to search a binary tree are discussed in Chapter 12.

Do you use a binary search when you look up someone in a telephone directory? No, you typically don't open the phone book in the middle. Instead you estimate where in the book the name should be and open it to that point. For example, if you were looking for the phone number of Mildred Smedley, you would open the phone book closer to the end rather than at the middle. This search is called the *interpolation search*. The number of comparisons needed for the average interpolation search is  $O(\log_2(\log_2 N))$ . The difficult part of designing an interpolation search is determining how to calculate an estimate of a value's location. We leave that problem for a more advanced course.

## Hashing

So far, we have succeeded in paring down our  $O(N)$  search to  $O(\log_2 N)$  by keeping the list ordered sequentially with respect to the key field—that is, the key in the first element is less than (or equal to) the key in the second element, which is less than the key in the third, and so on. Can we do better than that? Is it possible to design a search of  $O(1)$ —that is, one that takes the same search time to find any element in the list?

That is not an impossible dream. Suppose we would like to keep a list of repair records for 100 microcomputers. Each microcomputer is assigned a unique ID number between 1 and 100. We can use the ID numbers to index an array. Here are the declarations:

```
type Repair_Rec is
 record
 .
 .
 .
 end record;

subtype ID_Type is Positive range 1..100;

type Repair_Array is array (ID_Type) of Repair_Rec;

Repair_List : Repair_Array;
```

To find the repair record for any microcomputer, we use its ID number to index the array. For example, the repair record for microcomputer 57 is `Repair_List (57)`. Finding the record is an  $O(1)$  search.

In our repair record example, there is a one-to-one correspondence between the element's key and the array index. In practice, however, this perfect relationship between the key value and the address of an element is not easy to establish or maintain. Consider a small company that uses its 100 employees' five-digit ID number as the primary key field. Now the range of key values is from 00000 to 99999. Obviously it's impractical to set up an array of 100,000 elements, of which only 100 are needed, just to make sure that each employee's element is in a perfectly unique and predictable location.

What if we keep the array size down to the size that we actually need (an array of 100 elements) and just use the last two digits of the key field to identify each employee? For instance, the element of employee 53374 is in `Employee_List (74)`, and the element of employee 81235 is in `Employee_List (35)`. Note that the elements are not ordered according to the *value* of the key field as they were in our earlier discussion; the position of Employee 81235's record precedes that of Employee 53374 in the array, even though the value of its key is larger. Instead, the elements are ordered with respect to some *function* of the key value.

---

#### Hash Function

A function used to manipulate the key of an element in a list to identify its location in the list.

#### Hashing

The technique used for ordering and accessing elements in a list in a relatively constant amount of time by manipulating the key to identify its location in the list.

---

This function is called a **hash function**, and the search technique we are using is called **hashing**. In the case of the employee list above, the hash function is `Key rem 100`. The key (`ID_Num`) is divided by 100, and the remainder is used as an index into the array of employee elements, as illustrated in Figure 15-1. This function assumes that the array is indexed from 0 to 99. The Ada function to perform the conversion of `ID_Num` values to indexes is very simple:

```
function Hash (ID_Num : in Positive) return Index_Type is

-- Turns 5-digit ID numbers into 2-digit index values

-- for an array that is indexed from 0 to 99

begin
 return ID_Num rem 100;
end Hash;
```

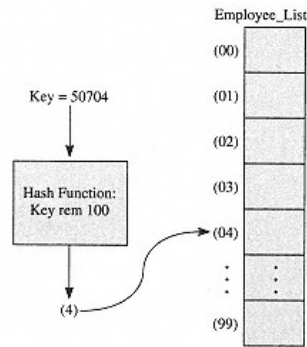


Figure 15-1  
Using a Hash Function to Determine the  
Location of the Element in an Array

This hash function has two uses. As we have seen, it's used as a method of accessing the element. The output of the hash function tells us where to *look* for a particular element, the information we need to Retrieve, Modify, or Delete the element. Here, for example, is a simple version of procedure `Retrieve`, which assumes that the element is in the list.

```
procedure Retrieve (List : in List_Type;
 Key : in Key_Type;
 Element : out Element_Type) is
-- Find element in List with Key, return copy in Element.

-- Preconditions: The element is in the list

-- Postconditions: Element is the item with Key

 Location : Index_Type;

begin -- Retrieve
 -- Use the hash function to find the element with Key
 Location := Hash (Key);
 -- Access the element through the hashed key
 Element := List.Info(Location);
end Retrieve;
```

There is a second use of the hash function. It determines where in the array to *store* the element. If the employee list elements were inserted into the list using the Insert operation from Chapter 8—into sequential array slots with their relative order determined by the Key field—we could not use the hash function to retrieve them. We have to create a version of the Insert operation that puts each new element into the correct slot *according to the hash function*. Here is a simple version of Insert, which assumes that the array slot at the index returned from the hash function is not in use:

```
procedure Insert (List : in out List_Type;
 New_Element : in Element_Type) is
-- Add New_Element to List
```



```
-- Preconditions: Slot Hash(Key_Of (New_Element)) is free

-- Postconditions: New_Element in List

Location : Index_Type;

begin -- Insert
 -- Use the hash function to find the insertion location
 Location := Hash(Key_Of (New_Element));
 -- Insert the new element into this array slot
 List.Info(Location) := Element;
 List.Length := List.Length + 1;
end Insert;
```

Figure 15-2(a) shows an array whose elements—records for the employees with the Key values (unique ID numbers) 12704, 31300, 49001, 52202, and 65606—were added using the `Insert` procedure above. Note that procedure `Insert` does not fill the array positions sequentially. Because we have not yet inserted any elements whose keys produce the hash values 3 and 5, the array slots (3) and (5) are logically "empty." This is different from the approach we used in Chapter 8 to create a key-ordered list. In Figure 15-2(b), the same employee records have been inserted into a key-ordered list using the `Insert` operation from Chapter 8. Note that, unless the hash function was used to determine where to insert an element, the hash function is *useless* for finding the element.

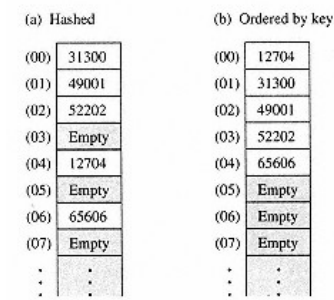


Figure 15-2  
Comparing Hashed and Key Ordered Lists of  
Identical Elements

Collisions

|                                                                           |
|---------------------------------------------------------------------------|
| <b>Collision</b>                                                          |
| Condition resulting when two or more keys produce the same hash location. |

By now you are probably objecting to this scheme on the grounds that it doesn't guarantee unique addresses. ID number 01234 and ID number 91234 both "hash" to the same address: `List.Info(34)`. The problem of avoiding these **collisions** is the biggest challenge in designing a good hash function. A good hash function *minimizes collisions* by spreading the elements uniformly throughout the array. We say "minimizes collisions," for it is extremely difficult to avoid them completely.

Assuming that there are to be some collisions, where do you store the elements that produce them? We briefly describe several popular collision-handling algorithms in the next sections. Note that the scheme that is used to find the place to store an element determines the method subsequently used to retrieve it.

## Linear Probing

Resolving a hash collision by sequentially searching a hash table beginning at the location returned by the hash function.

### Linear Probing

A simple approach to resolving collisions is to store the colliding element in the next available space. This technique is known as **linear probing**. In the situation in Figure 15-3, we want to add the employee element with the key ID number 77003. The hash function returns the `Location` (03). But there is already an element stored in this array slot, the record for Employee 50003. We increment `Location` to (04) and examine the next array slot. `List.Info(4)` is also in use, so we increment the index again. This time we find a slot that is empty, so we store the new element in `List.Info(5)`.

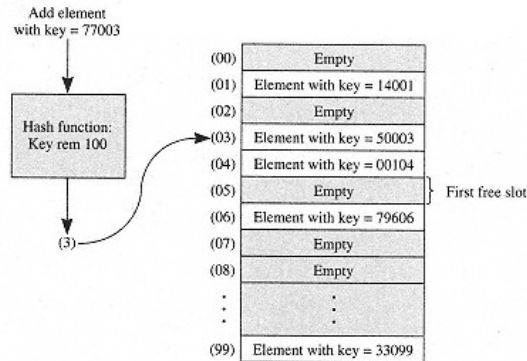


Figure 15-3  
Handling Collisions with Linear Probing

What happens if the key hashes to the last index in the array and that space is in use? We can consider the array as a circular structure and continue looking for an empty slot at the beginning of the array. This situation is similar to our circular array-based queue in Chapter 7. There, we used the `rem` operator when we incremented our index. We can use similar logic here.

How do we know whether an array slot is empty? We can include a parallel array, `Status`, to indicate the status of each slot in the list. Each element in this array is given an initial value of `Empty`. Here are the declarations for `List_Type` that includes the slot status array:

```
type Element_Array is array (Positive range <>) of Element_Type;

type Status_Type is (Empty, Used);
type Status_Array is array (Positive range <>) of Status_Type;

type List_Type is
 record
 Length : Natural := 0;
 Info : Element_Array (Index_Type);
 Status : Status_Array (Index_Type) := (Index_Type => Empty);
 end record;
```

We have retained the `Length` field of `List_Type`. Even if it doesn't tell us where the end of the list is, it's still useful in determining if the list is full. Here is a version of procedure `Insert` that uses linear probing to find a place to store a new element:

```

procedure Insert (List : List_Type;
 New_Element : Element_Type) is
-- Insert New_Element into the List using function Hash

-- Preconditions: None

-- Postconditions: New_Element is in List

-- Exceptions: OVERFLOW if there is no room remaining in List

 Location : Index_Type; -- Index returned by Hash function

begin -- Insert
 if List.Length = List.Info'Length then
 raise OVERFLOW;
 else
 -- Search for the first empty slot
 Location := Hash (Key_Of (New_Element));
 loop -- Each iteration, check a slot status
 exit when List.Status(Location) = Empty;
 -- Increment Location handling wrap-around from last to first.
 Location := (Location + 1) rem List.Info'Length;
 end loop;
 -- Insert the new element into this slot.
 List.Info(Location) := New_Element;
 List.Status(Location) := Used;
 List.Length := List.Length + 1;
 end if;
end Insert;

```

Note that to handle the wrap-around from the end of the array to the beginning, we `rem` by the size of the array (`List.Info'Length`) not by the number of elements currently in the list (`List.Length`).

To search for an element using this collision-handling technique, we perform the hash function on the key, then compare the desired key to the actual key in the element at the designated location. If the keys do not match, we use linear probing, beginning at the next slot in the array. Following is a version of procedure `Retrieve` that uses this approach. If the element is not found in the list, the exception `KEY_ERROR` is raised.

```

procedure Retrieve (List : in List_Type;
 Key : in Key_Type;
 Element : out Element_Type) is
-- Retrieve a copy of the Element with Key

-- Preconditions: None

-- Postconditions: Element is a copy of the element in List with Key

-- Exceptions: KEY_ERROR is raised if there is no element in List
-- with Key

```

```

Start_Location : Index_Type;
Location : Index_Type;
Searched_All : Boolean;

begin -- Retrieve
 -- Find likely Location using Hash function
 Start_Location := Hash (Key);
 -- Initialize for search
 Location := Start_Location;
 Searched_All := False;
 -- Search until we find the key, an empty spot,
 -- or the entire array has been searched
 loop
 exit when List.Status(Location) = Empty or else
 Key = Key_Of (List.Info(Location)) or else
 Searched_All;
 -- Increment location
 Location := (Location + 1) rem List.Info'Length;
 -- Have we come back to where we started?
 Searched_All := Location = Start_Location;
 end loop;
 if List.Status(Location) = Used and then
 Key = Key_Of (List.Info(Location)) then
 Element := List.Info(Location);
 else
 raise KEY_ERROR;
 end if;
end Retrieve;

```

We have discussed the insertion and retrieval of elements in a hash table, but we have not yet mentioned how to delete an element from the table. If we did not need to concern ourselves with collisions, the deletion algorithm would be simple:

### Delete

```

Location := Hash (Key)
List.Status(Location) := Empty

```

Collisions, however, complicate the matter. We can find the element using the same search approach as we used for `Retrieve`. But when we locate the element in the hash table, we can't merely set its status indicator to `Empty`. A review of the `Retrieve` procedure above shows the problem. In the loop, the detection of an empty slot ends the search. If procedure `Delete` "empties" the slot occupied by a deleted element, we may terminate a subsequent search prematurely.

Let's look at an example. In Figure 15-4, suppose we delete the element with the key 77003 by setting the status value of array slot (05) to `Empty`. A subsequent search for the element with the key 42504 would begin at the hash address (04). The record in this slot isn't the one we're looking for, so we increment the hash address to (05). The status of this slot, which formerly was occupied by the record that we deleted, is now `Empty`, so we terminate the search. We haven't really finished searching, however—the record that we're looking for is in the next slot.

|                     |      |                          |
|---------------------|------|--------------------------|
|                     | (00) | Empty                    |
|                     | (01) | Element with key = 14001 |
|                     | (02) | Empty                    |
| Order of Insertion: | (03) | Element with key = 50003 |
| 14001               | (04) | Element with key = 00104 |
| 00104               | (05) | Element with key = 77003 |
| 50003               | (06) | Element with key = 42504 |
| 77003               | (07) | Empty                    |
| 42504               | (08) | Empty                    |
| 33099               |      |                          |
| ⋮                   | ⋮    | ⋮                        |
| ⋮                   | ⋮    | ⋮                        |
|                     | (99) | Element with key = 33099 |

Figure 15-4  
A Hash Table with Linear Probing

One solution to this problem is to create a third status value, `Deleted`, to use in slots that were occupied by deleted records. A status of `Deleted` means that this slot is currently free, but the slot was previously occupied.

With this change, we must modify both the insertion and retrieval operations to process slots correctly. The insertion algorithm treats a slot that has status `Empty` or `Deleted` the same; the search for an available slot for the new element ends. A status of `Empty` halts the search in procedure `Retrieve`, but a status of `Deleted` does not.

This solution corrects the search problem, but generates another: after many deletions, the search "path" to a record may travel through many array slots with status `Deleted`. This may cause the efficiency of retrieving an element to deteriorate. These problems illustrate that hash tables, in the forms that we have studied thus far, are not the most effective data structure for implementing tables whose elements may be deleted.

---

**Clustering**  
The tendency of elements to become unevenly distributed in the hash table, with many elements clustering around a single address.

---

### Clustering

Another problem with linear probing is that it results in a situation called **clustering**. A good hash function results in a uniform distribution of indexes throughout the array's index range. Initially, therefore, records are inserted throughout the array, each slot equally likely to be filled. Over time, however, after a number of collisions have been resolved, the distribution of records in the array becomes less and less uniform. The records tend to cluster together, as multiple keys begin to compete for a single address.

Consider the hash table in Figure 15-4. Only a record whose key produces the hash address 8 would be inserted into array slot (08). However, any records with keys that produce the hash addresses 3, 4, 5, 6, or 7 would be inserted into array slot (07). That is, array slot (07) is five times as likely as array slot (08) to be filled. Clustering results in inconsistent efficiency of insertion and retrieval operations.

---

**Rehashing**  
Resolving a collision by computing a new hash location from a hash function that manipulates the original location rather than the element's key.

---

### Rehashing

The technique of linear probing discussed above is an example of collision resolution by **rehashing**. If the hash function produces a collision, the hash address is used as the input to a *rehash function* to compute a new address. In the previous section, we added 1 to the hash address to create a new hash address; that is, we used the rehash function:

```
(Hash_Address + 1) rem 100.
```

For rehashing with linear probing, you can use any function

$$(\text{Hash\_Address} + \text{constant}) \text{ rem array-size}$$

as long as *constant* and *array-size* are relatively prime—that is, if the largest number that divides both of them evenly is 1. For instance, given the 100-slot array in Figure 15-5, we might use the constant 3 in the rehash function:

$$(\text{Hash\_Address} + 3) \text{ rem } 100.$$

(Though 100 is not a prime number, 3 and 100 are relatively prime; they have no common factor larger than 1.)

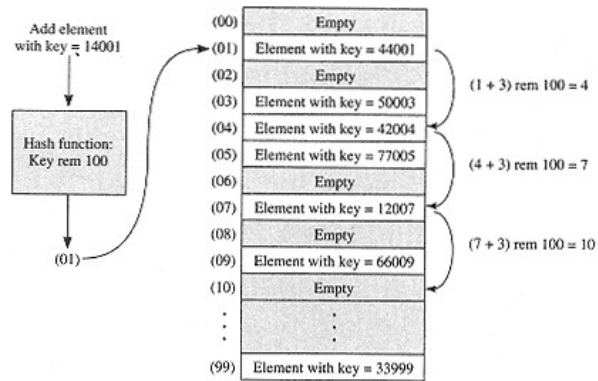


Figure 15-5  
Handling Collisions with Rehashing

Suppose that we want to add a record with the key 14001 to the hash table in Figure 15-5. The original hash function ( $\text{Key rem } 100$ ) returns the hash address (01), but this array slot is in use; it contains the record with the key 44001. To determine the next array slot to try, we apply the rehash function using the results of the first hash function as input:  $(1 + 3) \text{ rem } 100 = 4$ . The array slot at index (04) is also in use, so we reapply the rehash function until we get an available slot. Each time, we use the address computed from the previous rehash as input to the rehash function. The second rehash gives us  $(4 + 3) \text{ rem } 100 = 7$ ; this slot is in use. The third rehash gives us  $(7 + 3) \text{ rem } 100 = 10$ ; the array slot at index (10) is empty, so the new element is inserted there.

To understand why the constant and the number of array slots must be relatively prime, consider the rehash function:

$$(\text{Hash\_Address} + 2) \text{ rem } 100.$$

We want to add the record with the key 14001 to the hash table pictured in Figure 15-5. The original hash function,  $\text{Key rem } 100$ , returns the hash address (01). This array slot is already occupied. We resolve the collision by applying the rehash function above, examining successive odd-numbered indexes until a free slot is found. What happens if *all* of the

slots with odd-numbered indexes are already in use? The search would fail—even though there are free slots with even-numbered indexes. This rehash function does not cover the full index range of the array. However, if the constant and the number of array slots are relatively prime (like 3 and 100), the function produces successive rehashes that eventually cover *every* index in the array.

Rehash functions that use linear probing don't eliminate clustering (although the clusters aren't always visually apparent in a figure). For example, in Figure 15-5, any record with a key value that produces the hash address 1, 4, 7, or 10 would be inserted in the slot at index (10).

**Quadratic Probing**

Resolving a collision by applying a rehash function of the form  $(\text{Hash\_Address} + I^2) \bmod \text{Array\_Size}$ , where  $I$  is the number of times that the rehash function has been applied.

In linear probing, we add a constant (usually 1) in each successive application of the rehash function. Another approach, called **quadratic probing**, makes the result of rehashing dependent on how many times the rehash function has been applied. In the  $I$ th rehash, the function is

$$(\text{Hash\_Address} + I^2) \bmod \text{array-size}.$$

The first rehash adds 1 to the Hash\_Address, the second rehash adds 4, the third rehash adds 9, and so on. Quadratic probing reduces clustering, but it doesn't necessarily examine every slot in the array. For example, if *array-size* is a power of 2 (512 or 1024, for example), relatively few array slots are examined. However, if *array-size* is a prime number of the form  $(4 * \text{some-integer} + 3)$ , quadratic probing does examine every slot in the array.

**Random Probing**

Resolving a collision by reaching random locations in a hash table for the hash target.

A third approach uses a pseudorandom number generator (such as the one in Appendix K) to determine the increment to Hash\_Address in each application of the rehash function. **Random probing** is excellent for eliminating clustering, but it tends to be slower than the other techniques we have discussed.

**Bucket**

A collection of records associated with a particular hash address.

### Buckets and Chaining

Another alternative for handling collisions is to *allow* multiple element keys to hash to the same location. One solution is to let each computed hash address contain slots for multiple elements, rather than just a single element. Each of these multi-element locations is called a **bucket**. Figure 15-6 shows a hash table with buckets that can contain three elements each. Using this approach, we can allow collisions to produce duplicate entries at the same hash address, up to a point. When the bucket becomes full, we must again deal with handling collisions.

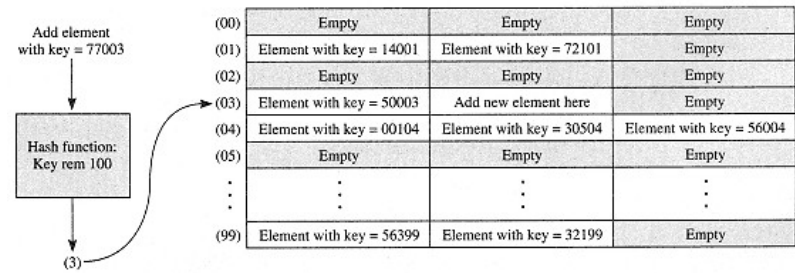


Figure 15-6  
Handling Collisions by Hashing with Buckets

**Chain**

Linked list of elements that share the same hash address.

Another solution, which avoids this problem, is to use the hash address not as the actual location of the element, but as the index into an array of pointers. Each pointer accesses a **chain** of elements that share the same hash address. Figure 15-7 illustrates this solution to the problem of collisions. Rather than rehashing, we simply allow both elements to share hash address (03). The entry in the array at this location contains a pointer to a linked list that includes both elements.

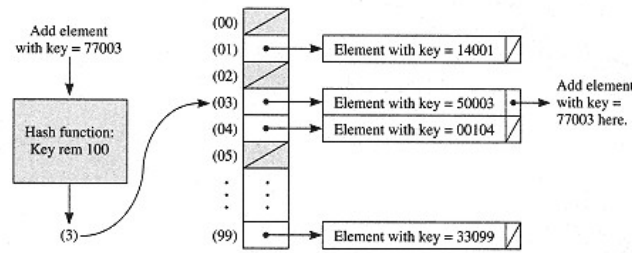


Figure 15-7  
Handling Collisions by Hashing with Chaining

To search for a given element, you first apply the hash function to the key and then search the chain for the element. Searching isn't eliminated, but it's limited to elements that actually share a hash address. Using the first hash-and-search technique discussed, you may have to search through many additional elements if the slots following the hash address are filled with elements from collisions on other addresses.

Figure 15-8 illustrates a comparison of the chaining and hash-and-search schemes. The elements were added in the following order:

45300  
20006  
50002  
40000  
25001  
13000  
65905  
30001  
95000

Figure 15-8(a) represents the linear probing approach to collision handling; Figure 15-8(b) shows the result of chaining the colliding elements. Let's search for the element with the key 30001.

Using linear probing, we apply the hash function to get the index (1). Because `List.Info(1)` does not contain the element with the key 30001, we search sequentially until we find the element in `List.Info(7)`.

Using the chaining approach, we apply the hash function to get the index (1). `List.Info(1)` directs us to a chain of elements whose keys hash to 1. We search this linked list until we find the element with the desired key. Another advantage of chaining is



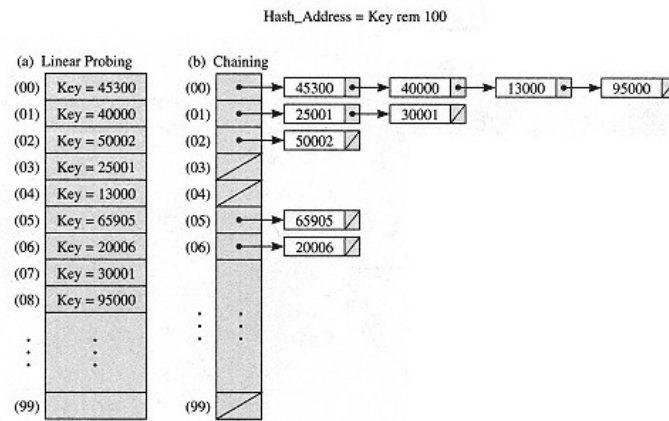


Figure 15-8  
Comparison of Linear Probing and Chaining Schemes

that it simplifies the deletion of records from the hash table. We apply the hash function to obtain the index of the array slot that contains the pointer to the appropriate chain. The node can then be deleted from this chain using the linked list algorithm from Chapter 8.

In fact, we can use the linked list implementation of the sorted list class developed in Chapter 8 directly in the declaration of our hash table.

```
package Employee_List is new Sorted_List (Element_Type => ...);
```

```
type Hash_Table is array (0..99) of Employee_List.List_Type;
```

Each entry in the hash table is a sorted linked list—a chain of elements that hash to the same location. We use our hash function to find the appropriate chain and then the sorted list operations to retrieve, modify, insert, or delete the element from the list.

### Choosing a Good Hash Function

One way to minimize collisions is to use a data structure that has more space than is actually needed for the number of elements, in order to increase the range of the hash function. In practice it's desirable to have the array size somewhat larger than the number of elements required in order to reduce the number of collisions.

Selecting the table size involves a space versus time trade-off. The larger the range of hash addresses, the less likely it is that two keys hash to the same location. However, allocating an array that contains a large number of empty spaces wastes space. An array size that is two times the anticipated number of elements usually provides adequate results.

More important, you can design your hash function to minimize collisions. The goal is to distribute the elements as uniformly as possible throughout the array. Therefore, you want your hash function to produce unique addresses as often as possible. Once you admit collisions, you must introduce some sort of searching, either through array or chain searching or through rehashing. The access to each element is no longer direct, and the search is

no longer  $O(1)$ . In fact, if the collisions cause very disproportionate chains, the worst case may be almost  $O(N)$ !

To avoid such a situation, you need to know something about the distribution of keys. Imagine a company whose employee elements are ordered according to a company ID six digits long. There are 500 employees, and we decide to use a chained approach to handling collisions. We set up 100 chains (expecting an average of five elements per chain) and use the hash function

```
ID_Num rem 100
```

That is, we use the last two digits of the six-digit ID number as our index. The planned hash scheme is shown in Figure 15-9(a).

Figure 15-9(b) shows what happened when the hash scheme was implemented. How could the distribution of the elements have come out so skewed? It turns out that the company's ID number is a concatenation of three fields:

|                                         |                                   |                                       |
|-----------------------------------------|-----------------------------------|---------------------------------------|
| <u>X X X</u>                            | <u>X</u>                          | <u>X X</u>                            |
| 3 digits,<br>unique number<br>(000–999) | 1 digit,<br>dept. number<br>(0–9) | 2 digits,<br>year hired<br>(e.g., 93) |

The hash scheme depended solely on the year hired to produce unique addresses. Because the company was founded in 1993, all the elements were crowded very disproportionately into a small subset of the hash addresses. A search for an employee element, in this case, is  $O(N)$ . Although this is an exaggerated example, it illustrates the need to understand as completely as possible the domain and predicted values of keys in a hash scheme.

Let's see if we can find a hash function that distributes the employees more evenly in our hash table. The first three digits of the company ID number provide a unique value for each employee. Let's take the two rightmost digits from this group of three digits. The following hash function does the job.

```
(ID_Num / 1000) rem 100
```

### Division Method

The most common hash functions use the division method (`rem`) to compute hash addresses. This is the type of function used in the preceding examples. The general function is

```
Key rem Table_Size
```

(or `Key rem Table_Size + 1` to index the table beginning at 1, rather than at 0). We have already mentioned the idea of making the table somewhat larger than the number of elements requires, in order to increase the range of hash addresses. In addition, it has been found that better results are produced with the division method when the table size is prime.

The advantage of the division hash function is simplicity. Sometimes, however, it's necessary to use a more complicated (or even exotic) hash function to get a good distribution of hash addresses.

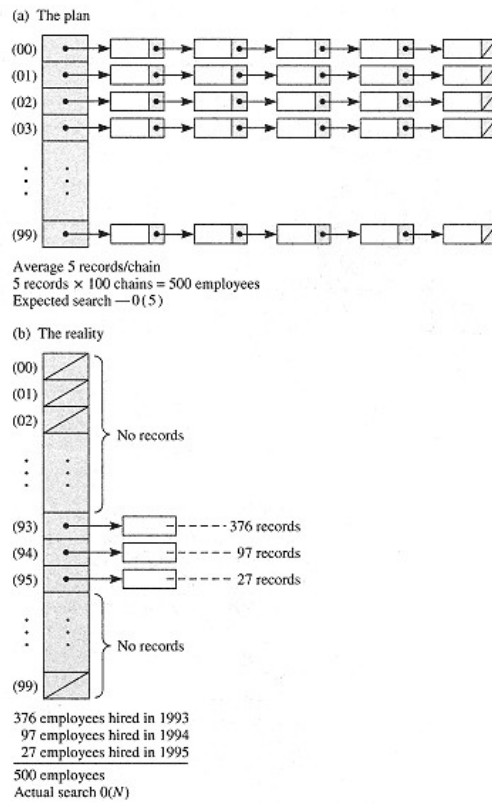


Figure 15-9  
 Hash Scheme to Handle Employee Elements

### Other Hash Methods

How can we use hashing if the key field is a string instead of an integer? One approach is to use the ordinal values (`'Pos`) of the string's characters to create a number that can be used as an index. For instance, here is a simple hash function that converts a key of type `String` to an index:

```
function Hash (Key : in String) return Index_Type is
-- Converts String Key to an array index by dividing the sum of the
-- ordinal values by the hash table size. The remainder is the index.
```

```
 Sum : Natural;
```

```
begin -- Hash
 Sum := 0;
```

```

for Index in Key'Range loop
 Sum := Sum + Character'Pos (Key(Index));
end loop;
return Sum rem Table_Size;
end Hash;

```

---

**Folding**

A hash method that breaks the key into several pieces and concatenates or exclusive-ORs some of them to form the hash address.

---

A hash method called **folding** involves breaking the key into several pieces and concatenating or exclusive-ORing some of them to form the hash address. Another method is to square the key and then use some of the digits (or bits) of the key as a hash address. There are a number of other techniques, all of which are intended to make the hash address as unique and random (within the allowed range) as possible.

Let's look at an example of folding in a hash function. Suppose we want to devise a hash function that results in an index between 0 and 255, and the internal representation of the Integer-type key is a bit string of 32 bits. We know that it takes 8 bits to represent the 256 index values ( $2^8 = 256$ ). A folding algorithm to create a hash function might

1. break the key into four bit strings of 8 bits each,
2. exclusive-OR the first and last bit strings,
3. exclusive-OR the two middle bit strings, and
4. exclusive-OR the results of steps 2 and 3 to produce the 8-bit index into the array.

We illustrate this scheme using the key 618403. The binary representation of this key is

```
00000000000010010110111110100011.
```

We break this bit string into four 8-bit strings:

```
00000000 (leftmost 8 bits)
```

```
00001001 (next 8 bits)
```

```
01101111 (next 8 bits)
```

```
10100011 (rightmost 8 bits)
```

The next step is to exclusive-OR the first and last bit strings. (The exclusive-OR of two bits is 0 if the two bits are the same, and 1 if they are different. To exclusive-OR [denoted as `xor`] bit strings, we apply this rule to successive pairs of bits.)

```

 00000000
(xor) 10100011
 10100011

```

Then we exclusive-OR the middle 2-bit strings:

```

 00001001
(xor) 01101111
 01100110

```

Finally we exclusive-OR the results of the preceding two steps:

```

 10100011
(xor) 01100110

 11000101

```

This binary number is equivalent to the decimal number 197. So the key 618403 hashes into the index 197.

The relationship between the key and the index is not intuitively obvious, but the indexes produced are likely to be uniformly distributed through the range of possible values.

Let's look at how we might carry out these operations in Ada. We can use an array of Boolean values to store the bit-strings:

```

type Bit_String is array (Natural range <>) of Boolean;
-- A bit string with 32 bits
subtype Bit32 is Bit_String (0..31);

```

Ada's logical operators (`and`, `or`, `not`, and `xor`) can be used with both Boolean values and arrays of Boolean values. So combining bit strings with `xor` is easy.

But how can we convert a number into a bit string? One way is to use integer arithmetic much like we did in the function `Digit_Of` that we wrote for a radix sort instantiation in Chapter 14. Except here we deal with the binary representation of a number rather than its decimal representation. Here is an Ada function to convert a non-negative integer into a bit string:

```

function To_Bit_String (Value : in Natural) return Bit32 is

-- Converts a non-negative integer into it corresponding bit string

-- Precondition: Value < 2 ** 32

 Num : Natural; -- Local copy of Value
 Result : Bit32;

begin
 -- To Bit String
 Num := Value;
 for Index in Result'Range loop
 -- True is assigned if the least significant bit of Num is 1.
 -- The least significant bit is 1 for all odd numbers.
 Result(Index) := Num rem 2 = 1;
 -- Shift all the bits to the right one place
 Num := Num / 2;
 end loop;
 return Result;
end To_Bit_String;

```

Function `To_Bit_String` uses a loop to test all of the bits in the number. The rightmost bit of a number is the units position. All odd numbers have a 1 in this position. Each iteration, we test the least significant bit and then divide by two to shift all the bits to the right. This is equivalent to dividing a decimal number by 10 to shift all of the digits to the right. In both cases, the least significant bit or digit is discarded by the division.

To convert a bit string to a number we use our knowledge of the "value" of different positions in a number. For example, the decimal number 636 has six ones, three tens, and six hundreds. The positions in a decimal number are powers of 10:  $10^0$ ,  $10^1$ ,  $10^2$ ,  $10^3$ , and so on. In a binary number the positions are powers of 2:  $2^0$ ,  $2^1$ ,  $2^2$ ,  $2^3$ , and so on. This information is used in the following conversion function:

```
function To_Natural (Bits : in Bit_String) return Natural is
-- Converts bit string into a non-negative integer

 Result : Natural;

begin -- To Natural
 Result := 0;
 for Column in Bits'Range loop
 if Bits (Column) then -- Is there a 1 in this column?
 Result := Result + 2 ** Column; -- Add the column to the Result
 end if;
 end loop;
 return Result;
end To_Natural;
```

Both of these conversion functions contain a loop. We can use Ada's predefined generic function `Unchecked_Conversion` to carry out these conversions directly, without loops. However, you need a detailed knowledge of the representation of numbers used by *your particular computer* to use this function successfully. Programs that use `Unchecked_Conversion` are not usually *portable*; they can't be run on other computers without making changes to the program.

Now that we can convert between numbers and bit strings, we are ready to write a hash function to implement the folding algorithm outlined above. We use array slices to access the four 8-bit portions of our 32-bit bit string:

```
function Hash (Key : in Natural) return Index_Type is
-- This function uses folding to convert a numeric key
-- into an index (0 to 255).

 subtype Bit8 is Bit_String (0..7); -- An 8-bit bit string

 Bits : Bit32;
 Bit_Result : Bit8;
 Num_Result : Index_Type;

begin -- Hash
 Bits := To_Bit_String (Key); -- Convert Key to a 32-bit bit string
 -- Fold the bits into an 8-bit result
 Bit_Result := (Bits(0..7) xor Bits(24..31)) -- First and Last
 xor
 (Bits(8..15) xor Bits(16..23)); -- Two middle
 Num_Result := To_Natural (Bit_Result);
 return Num_Result;
end Hash;
```

When using an exotic hash function, you should keep two considerations in mind. First, the efficiency of calculating the function should be considered. Even if a hash function always produces unique addresses, it isn't a good hash function if it takes longer to calculate the address than to search half the list. Second, programmer time should be considered. An extremely exotic function that somehow produces unique addresses for all of the known key values may fail if the domain of possible key values changes in a later modification. The programmer who has to modify the program may then waste a lot of time trying to find another hash function that is just as clever.

Finally, we should mention that if you know all of the possible keys ahead of time, it's possible to determine a *perfect* hash function. For example, if we needed a list of elements whose keys were the 63 Ada reserved words (Appendix A), it's possible to find a hash function that hashes each word to a unique location in an array of 63 components. In general, it takes a great deal of effort to discover a perfect hash function. And usually, we find that its computational complexity is very high, perhaps comparable to the effort required to do a binary search. There are problems, like the microcomputer repair list discussed earlier, where a perfect hash function is obvious and simple.

### ***A Hash-Based List Package***

By now you are very familiar with Ada's construct for encapsulating classes: the package. We have given a package declaration for *every* data structure discussed in previous chapters. Now it's your turn to design a package declaration! It should, of course, contain all of our customary list operations. Most of these operations should behave identically to those we developed in previous list packages. The notable exception is the Traverse operation. Because our list elements are not ordered by key, more effort is required to traverse the elements in order. Of course, we could define our operation as one that processes the elements in some order other than by key. We leave the design and implementation of a hash-based list as an exercise.

### **Summary**

Searching, like sorting, is a topic that is closely tied to the goal of efficiency. We speak of a sequential search as an  $O(N)$  search, because it may require up to  $N$  comparisons to locate an element. ( $N$  refers to the number of elements in the list.) Binary searches are considered to be  $O(\log_2 N)$  and are appropriate for arrays only if they are sorted. A binary search tree may be used to support binary searches on a linked structure. The goal of hashing is to produce a search that approaches  $O(1)$ . Because of collisions of hash addresses, some searching or rehashing is usually necessary. A good hash function minimizes collisions and distributes the elements randomly throughout the table. There is no single best hash function—determining the properties of a hashing solution (array size and hash function) is very application dependent.

You can find packages in public or private Ada repositories that implement every sorting and searching method we have discussed. Why then have we devoted the past two chapters to a discussion of well-known sorting and searching algorithms? First, it's important to be familiar with several of the basic sorting and searching techniques. These are tools that you use over and over again in a programming environment, and you need to know which ones are appropriate solutions to different problems. You may have to choose

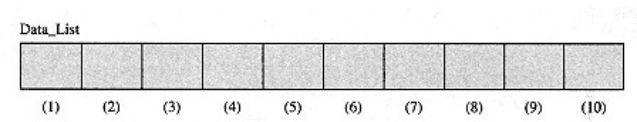
from several packages each using a different sorting or searching algorithm. To choose wisely, you need to understand the advantages and disadvantages of each method. Second, a review of sorting and searching techniques has given us another opportunity to reexamine a measuring tool—the Big-O approximation—that helps us determine how much work is required by a particular algorithm. Both building and measuring tools are needed to construct sound program solutions.

Exercises

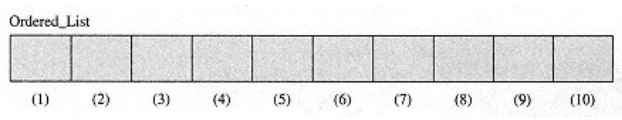
Exercises 1-4 use the following data file:

14 27 95 12 26 5 33 15 9 99

1. `Data_List` is an array of 10 integer positions. Show what `Data_List` would look like if it were loaded sequentially with the integers from the file (the first number in the first slot, the second in the second slot, and so on).



2. `Ordered_List` is an array of 10 integer positions. Show what `Ordered_List` would look like if it were loaded with the integers from the file and then the elements were sorted from smallest to largest.



3. `Tree` is a binary search tree of integer elements. Show what `Tree` would look like if the integers from the file were inserted into the tree in the order in which they were read into the file.

4. Fill in the following table, showing the number of comparisons needed either to find the value or to determine that the value is not in the list:

| Value | Search<br>Data_List<br>sequentially | Search<br>Ordered_List<br>sequentially | Binary search<br>Ordered_List | Search<br>Tree |
|-------|-------------------------------------|----------------------------------------|-------------------------------|----------------|
| 15    |                                     |                                        |                               |                |
| 17    |                                     |                                        |                               |                |
| 14    |                                     |                                        |                               |                |
| 5     |                                     |                                        |                               |                |
| 99    |                                     |                                        |                               |                |
| 100   |                                     |                                        |                               |                |
| 0     |                                     |                                        |                               |                |



Exercises 5-9 use the following values:

66 47 87 90 126 140 145 153 177 285 393 395 467 566 620 735

5. Store the values in a hash table with 20 positions, using the division method of hashing and the linear probing method of resolving collisions.
6. Store the values in a hash table with 20 positions, using rehashing as the method of collision resolution. Use  $\text{Key} \bmod \text{Table\_Size}$  as the hash function, and  $(\text{Key} + 3) \bmod \text{Table\_Size}$  as the rehash function.
7. Store the values in a hash table with 10 buckets, each containing three values. If a bucket is full, use the next (sequential) bucket that contains a free slot.
8. Store the values in a hash table that uses the hash function  $\text{Key} \bmod 10$  to determine which of 10 chains to put the value into.
9. Fill in the following table, showing the number of comparisons needed to find each value using the hashing representations given in Exercises 5-8.

| value | Problem 5 | Problem 6 | Problem 7 | Problem 8 |
|-------|-----------|-----------|-----------|-----------|
| 66    |           |           |           |           |
| 467   |           |           |           |           |
| 566   |           |           |           |           |
| 735   |           |           |           |           |
| 285   |           |           |           |           |
| 87    |           |           |           |           |

10. If you know the index of an element stored in an array of  $N$  unordered elements, which of the following best describes the order of the algorithm to retrieve the element?

- (a)  $O(1)$
- (b)  $O(N)$
- (c)  $O(\log_2 N)$
- (d)  $O(N^2)$
- (e)  $O(1/2N)$

11. The element being searched for is *not* in an array of 100 elements. What is the *average* number of comparisons needed in a sequential search to determine that the element is not there

- (a) if the elements are completely unordered?
- (b) if the elements are ordered from smallest to largest?
- (c) if the elements are ordered from largest to smallest?

12. The element being searched for is *not* in an array of 100 elements. What is the *maximum* number of comparisons needed in a sequential search to determine that the element is not there

- (a) if the elements are completely unordered?
- (b) if the elements are ordered from smallest to largest?
- (c) if the elements are ordered from largest to smallest?

13. The element being searched for is in an array of 100 elements. What is the *average* number of comparisons needed in a sequential search to determine the position of the element

- (a) if the elements are completely unordered?
- (b) if the elements are ordered from smallest to largest?
- (c) if the elements are ordered from largest to smallest?

14. Choose the answer that correctly completes the following sentence: The elements in an array may be ordered by highest probability of being requested in order to reduce
- (a) the average number of comparisons needed to find an element in the list.
  - (b) the maximum number of comparisons needed to detect that an element is not in the list.
  - (c) the average number of comparisons needed to detect that an element is not in the list.
  - (d) the maximum number of comparisons needed to find an element that is in the list.
15. *True or False?* If you answer False, correct the statement.
- (a) A binary search of an ordered set of elements in an array is always faster than a sequential search of the elements.
  - (b) A binary search is an  $O(N \log_2 N)$  algorithm.
  - (c) A binary search of elements in an array requires that the elements be sorted from smallest to largest.
  - (d) A high-probability ordering scheme would be a poor choice for arranging an array of elements that are equally likely to be requested.
  - (e) When a hash function is used to determine the placement of elements in an array, the order in which the elements are added does not affect the resulting array.
  - (f) When hashing is used, increasing the size of the array always reduces the number of collisions.
  - (g) If we use buckets in a hashing scheme, we don't have to worry about collision resolution.
  - (h) If we use chaining in a hashing scheme, we don't have to worry about collision resolution.
  - (i) The procedures in this chapter are used only for external searching.
  - (j) The goal of a successful hashing scheme is an  $O(1)$  search.
16. Choose the answer that correctly completes the following sentence: The number of comparisons required to find an element in a hash table with  $N$  buckets, of which  $M$  are full,
- (a) is always 1.
  - (b) is usually only slightly less than  $N$ .
  - (c) may be large if  $M$  is only slightly less than  $N$ .
  - (d) is approximately  $\log_2 M$ .
  - (e) is approximately  $\log_2 N$ .
17. A list is ordered in such a way that the elements most often accessed are closest to the top. The list is then searched sequentially to find the desired element. When it's found, it's moved toward the top of the list by swapping it with its immediate predecessor. Over time, the most frequently requested records migrate to the top of the list.
- (a) Show how the specifications of `Search_Array` (from Chapter 8) would be modified by this list representation.
  - (b) Assuming the list is implemented sequentially, using the declarations in Chapter 8, reimplement the `Search_Array` procedure.
  - (c) Assuming the list is implemented sequentially, using the declarations in Chapter 8, reimplement the `Insert` and `Delete` procedures.
18. A linked list is ordered in such a way that the elements most recently accessed are closest to the top. The list is searched sequentially to find the desired element. When it's found, it's moved to the beginning of the list.
- (a) Show how the specifications of `Search_Linked_List` (from Chapter 8) would be modified by this list representation.
  - (b) Assuming the list is implemented using the declarations of a linear linked list in Chapter 8, reimplement the `Search_Linked_List` procedure.
  - (c) Would this approach be good for a sequential list? Why or why not?
19. How might you order the elements in a list of Ada's reserved words to use the idea of high-probability ordering?
20. A record has a key field of type `String_Type [String (1..20)]`. A list of such records is stored in an array, using the following hash function: Each character in the string is turned into

its integer equivalent (using the `Character` attribute `'Pos`) and then the integers are added up to make one number, which is `remed` by `Table_Size`. The result is used as the index in the array.

Write function `Hash`, which takes the key string and returns the resulting index. (You may assume that all the characters in the packed array are legitimate; the string is padded with blanks to fill the array.)

21. Indicate whether each of the following changes would or would not be a good revision to function `Hash` from Exercise 20.

- (a) Instead of adding up the integer versions of all the characters, only add up the consonants.
- (b) Only add up the integer versions of the odd-indexed elements in the array.
- (c) Only add up the integer versions of the vowels.
- (d) Only use the integer version of the first letter in the string.
- (e) Only use the integer versions of the first five characters in the array.
- (f) Only use the integer versions of the last five characters in the array.
- (g) Instead of adding up the integer versions of all the letters, multiply them.

22. The chapter shows that the folding method hashed the key 618403 to the index 197.

- (a) What is another key between 32768 and 65536 that hashes to this index?
- (b) What is another key between 16384 and 32768 that hashes to this index?

23. The folding method developed in this chapter used the `xor` operator to combine several pieces. Why can't we use the normal `or` operator?

Use the following hash table for Exercises 24-26.

Seven elements have been inserted, using a hash function to determine the insertion place. The linear probing method of collision resolution has been used.

|      |    |
|------|----|
| (0)  | 51 |
| (1)  | 34 |
| (2)  |    |
| (3)  | 20 |
| (4)  | 3  |
| (5)  | 4  |
| (6)  | 38 |
| (7)  | 7  |
| (8)  |    |
| (9)  |    |
| (10) |    |
| (11) | 28 |
| (12) | 11 |
| (13) |    |
| (14) |    |
| (15) |    |
| (16) |    |

24. What is the hash function?
25. For each element in the hash table, show the result of the hash function and the number of places that had to be searched to find the insertion place.
26. Which of the following are possible insertion orders that would have produced the table above? (There may be more than one correct answer.)
- (a) 51 34 20 3 38 4 7 28 11
  - (b) 51 20 34 3 4 38 28 11 7
  - (c) 28 11 7 51 34 20 3 4 38
  - (d) 7 28 11 51 34 3 4 38 20
27. A hash table contains 1000 slots, indexed from 1 to 1000. The elements stored in the table have keys that range in value from 1 to 99,999. Which, if any, of the following hash functions would work correctly? (There may be more than one correct answer.)
- (a)  $\text{Key} \bmod 1000$
  - (b)  $(\text{Key} - 1) \bmod 1000$
  - (c)  $((\text{Key} + 1) \bmod 999)$
  - (d)  $(\text{Key} \bmod 1000) + 1$
28. A hash table contains 1000 slots, indexed from 0 to 999. The elements in the table have keys that range from 1 to 100,000. The original hash function is  $\text{Key} \bmod 1000$ . Which, if any, of the following collision resolution schemes would work correctly? (There may be more than one correct answer.)
- (a) Rehashing, with function  $= (\text{Key} + 1) \bmod 1000$ .
  - (b) Rehashing, with function  $= (\text{Key} + 2) \bmod 1000$ .
  - (c) Rehashing, with function  $= \text{Key} \bmod 999$ .
  - (d) Rehashing, with function  $= (\text{Key} + 3) \bmod 1000$ .
29. A hash table is used to store approximately 350 employee records of `Employee_Type`, whose key field is called `ID_Num`. The key field may range in value from 1000 to 9999. The hash table contains five buckets per hash address. The original hash function is  $\text{Key} \bmod 100$ .
- (a) Write the declarations for the hash table. Be sure to include initial values so the hash table is empty when it is elaborated.
  - (b) Write procedure `Insert`, which inserts an employee record into the first free slot in the appropriate bucket. (If the bucket is full, search the following buckets until a free slot is available.)
  - (c) Write procedure `Retrieve`, which returns a copy of the employee record with a designated key.
30. A hash table is used to store employee records of `Employee_Type`, whose key field is called `ID_Num`. The key field may range in value from 1000 to 9999. The original hash function is  $\text{Key} \bmod 100$ . There is a chain of records for each hash address.
- (a) Write the declarations for the hash table.
  - (b) Write procedure `Insert`, which inserts an employee record into the appropriate chain.
  - (c) Write procedure `Retrieve`, which returns a copy of the employee record with a designated key.
  - (d) Write procedure `Delete`, which deletes the employee record from its chain.

### Programming Problems

1. The object of this assignment is twofold. First, you are to compare the relative performance of different searching algorithms on the same data set. Second, you are to compare the performance of the same algorithm on data sets of different sizes.

Your program should compare the following three search strategies:

1. linear search in an unordered list
2. linear search in an ordered list
3. binary search

You need to write each search procedure with a counter that keeps track of how many comparisons are made.

### ***Input***

Create a data set of 100 integers. Do 10 searches with each algorithm. Be sure the searches include values not in the list as well as those in the list.

Create a second data set made up of three different sets of data to be searched. The first set of data should have 6 values, the second 50, and the third 150. Run each routine with five searches within each of the three data sets.

### ***Output***

Do the following for each data set:

1. Echo print the input data.
2. Print, for each search value:
  - (a) the value being searched for
  - (b) the algorithm name, the success of the search, and the number of comparisons made

### ***Turn In***

1. program listing
2. program output listed above
2. Create a data set with 100 integer values. Use the division method of hashing to store the data values into hash tables with table sizes of 7,51, and 151. Use the linear probing method of collision resolution. Print out the tables after the data values have been stored. Search for 10 different values in each of the three hash tables, counting the number of comparisons necessary. Print out the number of comparisons necessary in each case, in tabular form.

### ***Turn In***

1. program listing
2. program output listed above
3. (a) Write and compile the specification for a generic hash-table-based list ADO package declaration. Choose generic formal parameters to make your package as general as possible. Carefully document your list operations.  
 (b) Design, write, and test the hash-table-based list ADO package body using linear probing for collision resolution.
4. (a) Write and compile the specification for a generic hash-table-based list ADO package declaration. Choose generic formal parameters to make your package as general as possible. Carefully document your list operations.  
 (b) Design, write, and test the hash-table-based list ADO package body using buckets of five elements and linear probing to the next open bucket for collision resolution.
5. (a) Write and compile the specification for a generic hash-table-based list ADO package declaration. Choose generic formal parameters to make your package as general as possible. Carefully document your list operations.

- (b) Design, write, and test the hash-table-based list ADO package body using chaining for collision resolution.
6. (a) Write and compile the specification for a generic hash-table-based list set ADT package declaration. Choose generic formal parameters to make your package as general as possible. Carefully document your list operations.
- (b) Design, write, and test the hash-table-based set ADT package body using chaining for collision resolution.
7. (a) A hash table can be stored in a direct access file as well as in an array. Modify the hashbased list ADO you wrote for Programming Problems 3, 4, or 5 so that the hash table is stored in a direct file rather than an array. Use linear probing for collision resolution.
- (b) Discuss how chaining could be used to resolve collisions for a hash table stored in a direct access file.
8. Write and compile the specification for an ADT version of the hash-table-based list package that you designed in Programming Problem 3. Include all necessary type in the private part of the package.
- (a) Use linear probing for collision resolution.
  - (b) Use buckets of five elements and linear probing to the next open bucket for collision resolution.
  - (c) Use chaining for collision resolution.

Appendixes\*

Appendix A—  
Ada Reserved Words

Ada has 63 words that are reserved for special purposes. User-declared identifiers may not be reserved words. Certain attributes (Appendix I) are identical to some reserved words.

abort  
abs  
accept  
access  
all  
and  
array  
at  
  
begin  
body  
  
case  
constant  
  
declare  
delay  
delta  
digits  
do  
  
else  
elsif  
end  
entry  
exception  
exit  
  
for  
function  
  
generic  
goto  
  
if  
in  
is  
  
limited  
loop  
  
mod  
  
new  
not  
null  
  
of  
or  
others  
out  
  
package  
pragma  
private  
procedure  
  
raise  
range  
record  
rem  
renames  
return  
reverse  
  
select  
separate  
subtype  
  
task  
terminate  
then  
type  
  
use  
  
when  
while  
with  
  
xor

---

\* Appendixes A, B, D, E, F, G, and I contain material from the Ada Language Reference manual, copyright 1980, 1982, 1983 by the U.S. Government as represented by the Under Secretary of Defense, Research and Engineering. All rights reserved.

## Appendix B— Ada Syntax (EBNF)

One of the oldest computer-oriented metalanguages is the *Backus-Naur Form (BNF)*, which is named for John Backus and Peter Naur, who developed it in 1960. The Ada Language Reference Manual uses a simple variant of BNF, called Extended Backus-Naur Form (EBNF) to define the syntax of Ada. These syntax definitions are written out using letters, numbers, and four special symbols. The special symbols are:

|                 |                                                                 |
|-----------------|-----------------------------------------------------------------|
| <code>:=</code> | Stands for "is defined as"                                      |
| <code>[]</code> | Used to enclose an optional item                                |
| <code>{}</code> | Used to enclose an item that may be repeated zero or more times |
| <code> </code>  | Stands for "or" (used to separate alternative items)            |

All other symbols in the definitions are for use in Ada programs.

**Boldface** items denote literals, that is, symbols or words that should be written exactly as shown. Lowercase words, called nonterminal symbols, are those further defined in a separate EBNF rule. Names that begin with an *italicized* part are equivalent to the name without the italicized part. The italicized part simply conveys additional meaning within the context of the definition. Think of the italicized part as a comment, not actually part of the definition.

Here is the complete syntax of Ada in EBNF. Entries are in alphabetical order.

### A

|                                     |                 |                                                                                                                                                                                        |
|-------------------------------------|-----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>abort_statement</code>        | <code>:=</code> | <b>abort</b> <i>task_name</i> {, <i>task_name</i> };                                                                                                                                   |
| <code>accept_alternative</code>     | <code>:=</code> | <code>accept_statement</code> [ <code>sequence_of_statements</code> ]                                                                                                                  |
| <code>accept_statement</code>       | <code>:=</code> | <b>accept</b> <i>entry_simple_name</i> [( <i>entry_index</i> )]<br>[ <i>formal_part</i> ] [ <b>do</b> <code>sequence_of_statements</code><br><b>end</b> [ <i>entry_simple_name</i> ]]; |
| <code>access_type_definition</code> | <code>:=</code> | <b>access</b> <i>subtype_indication</i>                                                                                                                                                |
| <code>actual_parameter</code>       | <code>:=</code> | <code>expression</code>   <i>variable_name</i>  <br><code>type_mark</code> ( <i>variable_name</i> )                                                                                    |
| <code>actual_parameter_part</code>  | <code>:=</code> | ( <code>parameter_association</code><br>{, <code>parameter_association</code> })                                                                                                       |
| <code>address_clause</code>         | <code>:=</code> | <b>for</b> <i>simple_name</i> <b>use at</b> <i>simple_expression</i> ;                                                                                                                 |
| <code>aggregate</code>              | <code>:=</code> | ( <code>component_association</code><br>{, <code>component_association</code> })                                                                                                       |
| <code>alignment_clause</code>       | <code>:=</code> | <b>at mod</b> <i>static_simple_expression</i> ;                                                                                                                                        |
| <code>allocator</code>              | <code>:=</code> | <b>new</b> <i>subtype_indication</i>   <b>new</b><br><i>qualified_expression</i>                                                                                                       |
| <code>argument_association</code>   | <code>:=</code> | [ <i>argument_identifier</i> =>] <i>name</i>  <br>[ <i>argument_identifier</i> =>] <i>expression</i>                                                                                   |
| <code>array_type_definition</code>  | <code>:=</code> | <code>unconstrained_array_definition</code>  <br><code>constrained_array_definition</code>                                                                                             |



assignment\_statement ::= variable\_name := expression;

attribute ::= prefix'attribute\_designator

attribute\_designator ::= simple\_name [(universal\_static\_expression)]

B

base ::= integer

based\_integer ::= extended\_digit {[underline] extended\_digit}

based\_literal ::= base # based\_integer [.based\_integer]  
# [exponent]

basic\_character ::= basic\_graphic\_character | format\_effector

basic\_declaration ::= deferred\_constant\_declaration |  
exception\_declaration |  
generic\_declaration |  
generic\_instantiation |  
number\_declaration |  
object\_declaration |  
package\_declaration |  
renaming\_declaration |  
subprogram\_declaration |  
subtype\_declaration |  
task\_declaration |  
type\_declaration |

basic\_declarative\_item ::= basic\_declaration | representation\_clause |  
use\_clause

basic\_graphic\_character ::= upper\_case\_letter |  
digit |  
special\_character |  
space\_character

binary\_adding\_operator ::= + | - | &

block\_statement ::= [block\_simple\_name:]  
  
[declare  
    declarative\_part]  
  
begin  
    sequence\_of\_statements  
  
[exception  
    exception\_handler  
    {exception\_handler}]  
  
end [block\_simple\_name];

body ::= proper\_body | body\_stub

body\_stub ::= subprogram\_specification is separate; |  
    package body package\_simple\_name is separate; |  
    task body task\_simple\_name is separate;

C

|                              |                                                                                                                      |                                                                                                                                   |
|------------------------------|----------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| case_statement               | ::=                                                                                                                  | <b>case</b> expression <b>is</b><br>case_statement_alternative<br>{case_statement_alternative}<br><b>end case</b> ;               |
| case_statement_alternative   | ::=                                                                                                                  | <b>when</b> choice {   choice } =><br>sequence_of_statements                                                                      |
| character_literal            | ::=                                                                                                                  | 'graphic_character'                                                                                                               |
| choice                       | ::=                                                                                                                  | <i>component_simple_name</i>  <br>discrete_range        <br>simple_expression      <br><b>others</b>                              |
| code_statement               | ::=                                                                                                                  | type_mark' <i>record</i> _aggregate;                                                                                              |
| compilation                  | ::=                                                                                                                  | {compilation_unit}                                                                                                                |
| compilation_unit             | ::=                                                                                                                  | context_clause<br>library_unit        <br>context_clause<br>secondary unit                                                        |
| component_association        | ::=                                                                                                                  | [choice {   choice} =>] expression                                                                                                |
| component_clause             | ::=                                                                                                                  | component_name <b>at</b><br>static_simple_expression <b>range</b><br>static_range;                                                |
| component_declaration        | ::=                                                                                                                  | identifier_list:<br>component_subtype_definition<br>[:= expression];                                                              |
| component_list               | ::= component_declaration {component_declaration}  <br>{component_declaration} variant_part        <br><b>null</b> ; |                                                                                                                                   |
| component_subtype_definition | ::=                                                                                                                  | subtype_indication                                                                                                                |
| compound_statement           | ::=                                                                                                                  | accept_statement  <br>block_statement  <br>case_statement  <br>if_statement  <br>loop_statement  <br>select_statement             |
| condition                    | ::=                                                                                                                  | <i>boolean_expression</i>                                                                                                         |
| conditional_entry_call       | ::=                                                                                                                  | <b>select</b><br>entry_call_statement<br>[sequence_of_statements]<br><b>else</b><br>sequence_of_statements<br><b>end select</b> ; |
| constrained_array_definition | ::=                                                                                                                  | <b>array</b> index_constraint <b>of</b><br><i>component_subtype_indication</i>                                                    |

|                |    |                                                                                                                                |
|----------------|----|--------------------------------------------------------------------------------------------------------------------------------|
| constraint     | := | discriminant_constraint  <br>fixed_point_constraint  <br>floating_point_constraint  <br>index_constraint  <br>range_constraint |
| context_clause | := | {with_clause {use_clause}}                                                                                                     |

D

|                               |    |                                                                                              |
|-------------------------------|----|----------------------------------------------------------------------------------------------|
| decimal_literal               | := | integer [.integer] [exponent]                                                                |
| declarative_part              | := | {basic_declarative_item}<br>{later_declarative_item}                                         |
| deferred_constant_declaration | := | identifier_list : <b>constant</b> type_mark;                                                 |
| delay_alternative             | := | delay_statement<br>[sequence_of_statements]                                                  |
| delay_statement               | := | <b>delay</b> simple_expression;                                                              |
| derived_type_definition       | := | <b>new</b> subtype_indication                                                                |
| designator                    | := | identifier   operator_symbol                                                                 |
| discrete_range                | := | discrete_subtype_indication   range                                                          |
| discriminant_association      | := | [ <i>discriminant_simple_name</i><br>{   <i>discriminant_simple_name</i> } =>]<br>expression |
| discriminant_constraint       | := | (discriminant_association<br>{, discriminant_association})                                   |
| discriminant_part             | := | (discriminant_specification<br>{; discriminant_specification})                               |
| discriminant_specification    | := | identifier_list : type_mark<br>[:= expression]                                               |

E

|                                   |    |                                                                              |
|-----------------------------------|----|------------------------------------------------------------------------------|
| entry_call_statement              | := | <i>entry_name</i> [actual_parameter_part];                                   |
| entry_declaration                 | := | <b>entry</b> identifier [(discrete_range)<br>[formal_part];                  |
| entry_index                       | := | expression                                                                   |
| enumeration_literal               | := | identifier   character_literal                                               |
| enumeration_literal_specification | := | enumeration_literal                                                          |
| enumeration_representation_clause | := | <b>for</b> type_simple_name <b>use</b><br>aggregate;                         |
| enumeration_type_definition       | := | (enumeration_literal_specification<br>{, enumeration_literal_specification}) |
| exception_choice                  | := | <i>exception_name</i>   <b>others</b>                                        |
| exception_declaration             | := | identifier_list : <b>exception</b> ;                                         |

|                   |     |                                                                                                                                                                                                  |
|-------------------|-----|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| exception_handler | ::= | <b>when</b> exception_choice {   exception_choice } =><br>sequence_of_statements                                                                                                                 |
| exit_statement    | ::= | <b>exit</b> [ <i>loop_name</i> ] [ <b>when</b> condition];                                                                                                                                       |
| exponent          | ::= | <b>E</b> [+] integer   <b>E</b> - integer                                                                                                                                                        |
| expression        | ::= | relation { <b>and</b> relation }  <br>relation { <b>and then</b> relation }  <br>relation { <b>or</b> relation }  <br>relation { <b>or else</b> relation }  <br>relation { <b>xor</b> relation } |
| extended_digit    | ::= | digit   letter                                                                                                                                                                                   |

## F

|                              |     |                                                                                                                                                                                                                      |
|------------------------------|-----|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| factor                       | ::= | primary [ <b>**</b> primary]  <br><b>abs</b> primary  <br><b>not</b> primary                                                                                                                                         |
| fixed_accuracy_definition    | ::= | <b>delta</b> <i>static_simple_expression</i>                                                                                                                                                                         |
| fixed_point_constraint       | ::= | fixed_accuracy_definition<br>[range_constraint]                                                                                                                                                                      |
| floating_accuracy_definition | ::= | <b>digits</b> <i>static_simple_expression</i>                                                                                                                                                                        |
| floating_point_constraint    | ::= | floating_accuracy_definition<br>[range_constraint]                                                                                                                                                                   |
| formal_parameter             | ::= | <i>parameter_simple_name</i>                                                                                                                                                                                         |
| formal_part                  | ::= | (parameter_specification<br>{ ; parameter_specification })                                                                                                                                                           |
| format_effector              | ::= | <i>ASCII_horizontal_tabulation_character</i>  <br><i>ASCII_vertical_tabulation_character</i>  <br><i>ASCII_carriage_return_character</i>  <br><i>ASCII_line_feed_character</i>  <br><i>ASCII_form_feed_character</i> |
| full_type_declaration        | ::= | <b>type</b> identifier [discriminant_part] <b>is</b><br>type_definition;                                                                                                                                             |
| function_call                | ::= | <i>function_name</i> [actual_parameter_part]                                                                                                                                                                         |

## G

|                          |     |                                                                                                        |
|--------------------------|-----|--------------------------------------------------------------------------------------------------------|
| generic_actual_parameter | ::= | <i>entry_name</i>  <br>expression  <br><i>subprogram_name</i>  <br>type_mark  <br><i>variable_name</i> |
|--------------------------|-----|--------------------------------------------------------------------------------------------------------|

generic\_actual\_part ::= (generic\_association  
                           {, generic\_association})

generic\_association ::= [generic\_formal\_parameter =>]  
                           generic\_actual\_parameter

generic\_declaration ::= generic\_specification;

generic\_formal\_parameter ::= *parameter\_simple\_name* | operator\_symbol

generic\_formal\_part ::= **generic**  
                           {generic\_parameter\_declaration}

generic\_instantiation ::=

**package** identifier **is new** generic\_package\_name [generic\_actual\_part]; |  
     **procedure** identifier **is new** generic\_procedure\_name [generic\_actual\_part]; |  
     **function** designator **is new** generic\_function\_name [generic\_actual\_part];

generic\_parameter\_declaration ::=

    identifier\_list : [**in** | **out**] type\_mark [:= expression]; |  
     **type** identifier is generic\_type\_definition; |  
     private\_type\_declaration |  
     **with** subprogram\_specification [**is** name]; |  
     **with** subprogram\_specification [**is** <>];

generic\_specification ::= generic\_formal\_part  
                           subprogram\_specification |  
                           generic\_formal\_part  
                           package\_specification

generic\_type\_definition ::= (<>) |  
                           access\_type\_definition |  
                           array\_type\_definition |  
                           **delta** <> |  
                           **digits** <> |  
                           **range** <>

goto\_statement ::= **goto** *label\_name*;

graphic\_character ::= basic\_graphic\_character |  
                           lower\_case\_letter |  
                           other\_special\_character

## H

highest\_precedence\_operator ::= \*\* | abs | not

## I

identifier ::= letter {[underline] letter\_or\_digit}

identifier\_list ::= identifier {, identifier}

|                             |     |                                                                                                                                                                                          |
|-----------------------------|-----|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| if_statement                | ::= | <b>if</b> condition <b>then</b><br>sequence_of_statements<br><b>{elsif</b> condition <b>then</b><br>sequence_of_statements}<br><b>[else</b><br>sequence_of_statements]<br><b>end if;</b> |
| incomplete_type_declaration | ::= | <b>type</b> identifier [discriminant_part];                                                                                                                                              |
| index_constraint            | ::= | (discrete_range {, discrete_range})                                                                                                                                                      |
| index_subtype_definition    | ::= | type_mark <b>range</b> <>                                                                                                                                                                |
| indexed_component           | ::= | prefix(expression {, expression})                                                                                                                                                        |
| integer                     | ::= | digit {[underline] digit}                                                                                                                                                                |
| integer_type_definition     | ::= | range_constraint                                                                                                                                                                         |
| iteration_scheme            | ::= | <b>while</b> condition  <br><b>for</b> loop_parameter_specification                                                                                                                      |

L

|                              |     |                                                                                                                                                     |
|------------------------------|-----|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| label                        | ::= | <<label_simple_name>>                                                                                                                               |
| later_declarative_item       | ::= | body  <br>generic_declaration  <br>generic_instantiation  <br>package_declaration  <br>subprogram_declaration  <br>task_declaration  <br>use_clause |
| length_clause                | ::= | <b>for</b> attribute <b>use</b> simple_expression;                                                                                                  |
| letter                       | ::= | upper_case_letter   lower_case_letter                                                                                                               |
| letter_or_digit              | ::= | letter   digit                                                                                                                                      |
| library_unit                 | ::= | generic_declaration  <br>generic_instantiation  <br>package_declaration  <br>subprogram_body  <br>subprogram_declaration                            |
| library_unit_body:           | ::= | subprogram_body   package_body                                                                                                                      |
| logical_operator             | ::= | <b>and</b>   <b>or</b>   <b>xor</b>                                                                                                                 |
| loop_parameter_specification | ::= | identifier <b>in</b> [ <b>reverse</b> ] discrete_range                                                                                              |
| loop_statement               | ::= | [loop_simple_name:]<br>[iteration_scheme] <b>loop</b><br>sequence_of_statements<br><b>end</b> loop [loop_simple_name];                              |

**M**

mode ::= **[in] | in out | out**

multiplying\_operator ::= \* | / | **mod** | **rem**

**N**

name := attribute |  
character\_literal |  
indexed\_component |  
operator\_symbol |  
selected\_component |  
simple\_name |  
slice

null\_statement ::= **null**;

number\_declaration ::= identifier\_list : **constant** :=  
*universal\_static\_expression*;

numeric\_literal ::= decimal\_literal | based\_literal

**O**

object\_declaration ::=

    identifier\_list : [**constant**] subtype\_indication [:= expression]; |

    identifier\_list : [**constant**] constrained\_array\_definition [:= expression];

operator\_symbol ::= string\_literal

**P**

package\_body ::= **package body** *package\_simple\_name* **is**  
    [declarative\_part]  
    **[begin**  
        sequence\_of\_statements  
    **[exception**  
        exception\_handler  
        {exception\_handler}]]  
    **end** [*package\_simple\_name*];

package\_declaration ::= package\_specification;

package\_specification ::= **package** identifier **is**  
    {basic\_declarative\_item}  
    **[private**  
        {basic\_declarative\_item}]  
    **end** [*package\_simple\_name*]

|                          |     |                                                                                                                                                                           |
|--------------------------|-----|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| parameter_association    | ::= | [formal_parameter =>] actual_parameter                                                                                                                                    |
| parameter_specification  | ::= | identifier_list : mode type_mark<br>[:= expression]                                                                                                                       |
| pragma                   | ::= | <b>pragma</b> identifier [(argument_association<br>{, argument_association})];                                                                                            |
| prefix                   | ::= | name   function_call                                                                                                                                                      |
| primary                  | ::= | aggregate  <br>allocator  <br>(expression)  <br>function_call  <br>name  <br>null  <br>numeric_literal  <br>qualified_expression  <br>string_literal  <br>type_conversion |
| private_type_declaration | ::= | <b>type</b> identifier [discriminant_part] <b>is</b><br><b>[limited] private</b> ;                                                                                        |
| procedure_call_statement | ::= | <i>procedure_name</i> [actual_parameter_part];                                                                                                                            |
| proper_body              | ::= | package_body  <br>subprogram_body  <br>task_body                                                                                                                          |

Q

|                      |     |                                                 |
|----------------------|-----|-------------------------------------------------|
| qualified_expression | ::= | type_mark'(expression)  <br>type_mark'aggregate |
|----------------------|-----|-------------------------------------------------|

R

|                              |     |                                                                                                                                |
|------------------------------|-----|--------------------------------------------------------------------------------------------------------------------------------|
| raise_statement              | ::= | <b>raise</b> [exception_name];                                                                                                 |
| range                        | ::= | <i>range_attribute</i>  <br>simple_expression .. simple_expression                                                             |
| range_constraint             | ::= | <b>range</b> range                                                                                                             |
| real_type_definition         | ::= | floating_point_constraint  <br>fixed_point_constraint                                                                          |
| record_representation_clause | ::= | <b>for</b> <i>type_simple_name</i> <b>use</b><br><b>record</b> [alignment_clause]<br>{component_clause}<br><b>end record</b> ; |
| record_type_definition       | ::= | <b>record</b><br>component_list<br><b>end record</b>                                                                           |



`relation ::= simple_expression [relational_operator simple_expression] |`  
`simple_expression [not] in range |`  
`simple_expression [not] in type_mark`

`relational_operator ::= = | /= | < | <= | > | >=`

`renaming_declaration ::= identifier : type_mark renames object_name; |`  
`identifier : exception renames exception_name; |`  
`package identifier renames package_name; |`  
`subprogram_specification renames`  
`subprogram_or_entry_name;`

`representation_clause ::= type_representation_clause | address_clause`

`return_statement ::= return [expression];`

## S

`secondary_unit ::= library_unit_body | subunit`

`select_alternative ::= [when condition =>]`  
`selective_wait_alternative`

`select_statement ::= conditional_entry_call |`  
`selective_wait |`  
`timed_entry_call`

`selected_component ::= prefix.selector`

`selective_wait ::= select`  
`select_alternative`  
`{or`  
`select_alternative}`  
`[else`  
`sequence_of_statements]`  
`end select;`

`selective_wait_alternative ::= accept_alternative |`  
`delay_alternative |`  
`terminate_alternative`

`selector ::= all |`  
`character_literal |`  
`operator_symbol |`  
`simple_name`

`sequence_of_statements ::= statement`  
`{statement}`

`simple_expression ::= [unary_adding_operator] term`  
`{binary_adding_operator term}`

`simple name ::= identifier`

|                          |    |                                                                                                                                                                                                                                                   |
|--------------------------|----|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| simple_statement         | := | abort_statement  <br>assignment_statement  <br>code_statement  <br>delay_statement  <br>entry_call_statement  <br>exit_statement  <br>goto_statement  <br>null_statement  <br>procedure_call_statement  <br>raise_statement  <br>return_statement |
| slice                    | := | prefix(discrete_range)                                                                                                                                                                                                                            |
| statement                | := | {label} simple_statement  <br>{label} compound_statement                                                                                                                                                                                          |
| string_literal           | := | "{graphic_character}"                                                                                                                                                                                                                             |
| subprogram_body          | := | subprogram_specification <b>is</b><br>[declarative_part]<br><b>begin</b><br>sequence_of_statements<br>[ <b>exception</b><br>exception_handler<br>{exception_handler}]<br><b>end</b> [designator];                                                 |
| subprogram_declaration   | := | subprogram_specification;                                                                                                                                                                                                                         |
| subprogram_specification | := | <b>procedure</b> identifier [formal_part]  <br><b>function</b> designator [formal_part] <b>return</b><br>type_mark                                                                                                                                |
| subtype_declaration      | := | <b>subtype</b> identifier <b>is</b> subtype_indication;                                                                                                                                                                                           |
| subtype_indication       | := | type_mark [constraint]                                                                                                                                                                                                                            |
| subunit                  | := | <b>separate</b> ( <i>parent_unit_name</i> ) proper_body                                                                                                                                                                                           |

T

|                    |    |                                                                                                                                                                                                                                  |
|--------------------|----|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| task_body          | := | <b>task body</b> <i>task_simple_name</i> <b>is</b><br>[declarative_part]<br><b>begin</b><br>sequence_of_statements<br>[ <b>exception</b><br>exception_handler<br>{exception_handler}]<br><b>end</b> [ <i>task_simple_name</i> ]; |
| task_declaration   | := | task_specification;                                                                                                                                                                                                              |
| task_specification | := | <b>task</b> [ <b>type</b> ] identifier [ <b>is</b><br>{entry_declaration}<br>{representation_clause}<br><b>end</b> [ <i>task_simple_name</i> ]]                                                                                  |

|                            |     |                                                                                                                                                                                        |
|----------------------------|-----|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| term                       | ::= | factor {multiplying_operator factor}                                                                                                                                                   |
| terminate_alternative      | ::= | <b>terminate</b> ;                                                                                                                                                                     |
| timed_entry_call           | ::= | <b>select</b><br>entry_call_statement<br>[sequence_of_statements]<br><b>or</b><br>delay_alternative<br><b>end select</b> ;                                                             |
| type_conversion            | ::= | type_mark(expression)                                                                                                                                                                  |
| type_declaration           | ::= | full_type_declaration<br>incomplete_type_declaration<br>private_type_declaration                                                                                                       |
| type_definition            | ::= | access_type_definition<br>array_type_definition<br>derived_type_definition<br>enumeration_type_definition<br>integer_type_definition<br>real_type_definition<br>record_type_definition |
| type_mark                  | ::= | type_name   subtype_name                                                                                                                                                               |
| type_representation_clause | ::= | enumeration_representation_clause  <br>length_clause  <br>record_representation_clause                                                                                                 |

U

|                                |     |                                                                                                                       |
|--------------------------------|-----|-----------------------------------------------------------------------------------------------------------------------|
| unary_adding_operator          | ::= | +   -                                                                                                                 |
| unconstrained_array_definition | ::= | <b>array</b> (index_subtype_definition {, index_subtype_definition}) <b>of</b><br><i>component_subtype_indication</i> |
| use_clause                     | ::= | <b>use</b> <i>package_name</i> {, <i>package_name</i> };                                                              |

V

|              |     |                                                                                             |
|--------------|-----|---------------------------------------------------------------------------------------------|
| variant      | ::= | <b>when</b> choice {   choice} =><br>component_list                                         |
| variant_part | ::= | <b>case</b> <i>discriminant_simple_name</i> is<br>variant<br>{variant}<br><b>end case</b> ; |

W

|             |     |                                                                   |
|-------------|-----|-------------------------------------------------------------------|
| with_clause | ::= | <b>with</b> <i>unit_simple_name</i> {, <i>unit_simple_name</i> }; |
|-------------|-----|-------------------------------------------------------------------|

Appendix C—  
ASCII Character Set

The following chart shows the ordering of the ASCII (American Standard Code for Information Interchange) character set. There are three columns for each character. The first column contains the ordinal number of the character. The second column contains the character. The graphic characters are enclosed in apostrophes and the control characters are not. The third column contains the name of the constant, if defined, for the character declared in the predefined package ASCII.

| <i>Ordinal<br/>Value</i> | <i>Character</i> | <i>Constant<br/>Name</i> | <i>Ordinal<br/>Value</i> | <i>Character</i> | <i>Constant<br/>Name</i> |
|--------------------------|------------------|--------------------------|--------------------------|------------------|--------------------------|
| 0                        | nul              | NUL                      | 31                       | us               | US                       |
| 1                        | soh              | SOH                      | 32                       | ' '              |                          |
| 2                        | stx              | STX                      | 33                       | '!'              | EXCLAM                   |
| 3                        | etx              | ETX                      | 34                       | '"'              | QUOTATION                |
| 4                        | eot              | EOT                      | 35                       | '#'              | SHARP                    |
| 5                        | enq              | ENQ                      | 36                       | '\$'             | DOLLAR                   |
| 6                        | ack              | ACK                      | 37                       | '%'              | PERCENT                  |
| 7                        | bel              | BEL                      | 38                       | '&'              | AMPERSAND                |
| 8                        | bs               | BS                       | 39                       | '.'              |                          |
| 9                        | ht               | HT                       | 40                       | '('              |                          |
| 10                       | lf               | LF                       | 41                       | ')'              |                          |
| 11                       | vt               | VT                       | 42                       | '*'              |                          |
| 12                       | ff               | FF                       | 43                       | '+'              |                          |
| 13                       | cr               | CR                       | 44                       | ','              |                          |
| 14                       | so               | SO                       | 45                       | '-'              |                          |
| 15                       | si               | SI                       | 46                       | ':'              |                          |
| 16                       | del              | DEL                      | 47                       | '/'              |                          |
| 17                       | dc1              | DC1                      | 48                       | '0'              |                          |
| 18                       | dc2              | DC2                      | 49                       | '1'              |                          |
| 19                       | dc3              | DC3                      | 50                       | '2'              |                          |
| 20                       | dc4              | DC4                      | 51                       | '3'              |                          |
| 21                       | nak              | NAK                      | 52                       | '4'              |                          |
| 22                       | syn              | SYN                      | 53                       | '5'              |                          |
| 23                       | etb              | ETB                      | 54                       | '6'              |                          |
| 24                       | can              | CAN                      | 55                       | '7'              |                          |
| 25                       | em               | EM                       | 56                       | '8'              |                          |
| 26                       | sub              | SUB                      | 57                       | '9'              |                          |
| 27                       | esc              | ESC                      | 58                       | ':'              | COLON                    |
| 28                       | fs               | FS                       | 59                       | ','              | SEMICOLON                |
| 29                       | gs               | GS                       | 60                       | '<'              |                          |
| 30                       | rs               | RS                       | 61                       | '='              |                          |

| <i>Ordinal Value</i> | <i>Character</i> | <i>Constant Name</i> | <i>Ordinal Value</i> | <i>Character</i> | <i>Constant Name</i> |
|----------------------|------------------|----------------------|----------------------|------------------|----------------------|
| 62                   | '>'              |                      | 95                   | '_'              | UNDERLINE            |
| 63                   | '?'              | QUERY                | 96                   | '`'              | GRAVE                |
| 64                   | '@'              | ATSIGN               | 97                   | 'a'              | LC_A                 |
| 65                   | 'A'              |                      | 98                   | 'b'              | LC_B                 |
| 66                   | 'B'              |                      | 99                   | 'c'              | LC_C                 |
| 67                   | 'C'              |                      | 100                  | 'd'              | LC_D                 |
| 68                   | 'D'              |                      | 101                  | 'e'              | LC_E                 |
| 69                   | 'E'              |                      | 102                  | 'f'              | LC_F                 |
| 70                   | 'F'              |                      | 103                  | 'g'              | LC_G                 |
| 71                   | 'G'              |                      | 104                  | 'h'              | LC_H                 |
| 72                   | 'H'              |                      | 105                  | 'i'              | LC_I                 |
| 73                   | 'I'              |                      | 106                  | 'j'              | LC_J                 |
| 74                   | 'J'              |                      | 107                  | 'k'              | LC_K                 |
| 75                   | 'K'              |                      | 108                  | 'l'              | LC_L                 |
| 76                   | 'L'              |                      | 109                  | 'm'              | LC_M                 |
| 77                   | 'M'              |                      | 110                  | 'n'              | LC_N                 |
| 78                   | 'N'              |                      | 111                  | 'o'              | LC_O                 |
| 79                   | 'O'              |                      | 112                  | 'p'              | LC_P                 |
| 80                   | 'P'              |                      | 113                  | 'q'              | LC_Q                 |
| 81                   | 'Q'              |                      | 114                  | 'r'              | LC_R                 |
| 82                   | 'R'              |                      | 115                  | 's'              | LC_S                 |
| 83                   | 'S'              |                      | 116                  | 't'              | LC_T                 |
| 84                   | 'T'              |                      | 117                  | 'u'              | LC_U                 |
| 85                   | 'U'              |                      | 118                  | 'v'              | LC_V                 |
| 86                   | 'V'              |                      | 119                  | 'w'              | LC_W                 |
| 87                   | 'W'              |                      | 120                  | 'x'              | LC_X                 |
| 88                   | 'X'              |                      | 121                  | 'y'              | LC_Y                 |
| 89                   | 'Y'              |                      | 122                  | 'z'              | LC_Z                 |
| 90                   | 'Z'              |                      | 123                  | '{'              | L_BRACE              |
| 91                   | '['              | L_BRACKET            | 124                  | ' '              | BAR                  |
| 92                   | '\'              | BACK_SLASH           | 125                  | '}'              | R_BRACE              |
| 93                   | ']'              | R_BRACKET            | 126                  | '~'              | TILDE                |
| 94                   | ' `'             | CIRCUMFLEX           | 127                  | del              | DEL                  |

## Appendix D— Specification of the Package Text\_IO

This appendix contains the specification of the predefined package Text\_IO. It is taken directly from the Ada *Language Reference Manual*. In this listing, reserved words are in lowercase. All other identifiers are uppercase.

```
with IO_EXCEPTIONS;
package TEXT_IO is

 type FILE_TYPE is limited private;

 type FILE_MODE is (IN_FILE, OUT_FILE);

 type COUNT is range 0 .. implementation defined;
 subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;
 UNBOUNDED : constant COUNT := 0; -- line and page length

 subtype FIELD is INTEGER range 0 .. implementation defined;
 subtype NUMBER_BASE is INTEGER range 2 .. 16;

 type TYPE_SET is (LOWER_CASE, UPPER_CASE);

 -- File Management

 procedure CREATE (FILE : in out FILE_TYPE;
 MODE : in FILE_MODE := OUT_FILE;
 NAME : in STRING := "";
 FORM : in STRING := "");

 procedure OPEN (FILE : in out FILE_TYPE;
 MODE : in FILE_MODE;
 NAME : in STRING;
 FORM : in STRING := "");

 procedure CLOSE (FILE : in out FILE_TYPE);
 procedure DELETE (FILE : in out FILE_TYPE);
 procedure RESET (FILE : in out FILE_TYPE; MODE : in FILE_MODE);
 procedure RESET (FILE : in out FILE_TYPE);

 function MODE (FILE : in FILE_TYPE) return FILE_MODE;
 function NAME (FILE : in FILE_TYPE) return STRING;
 function FORM (FILE : in FILE_TYPE) return STRING;

 function IS_OPEN(FILE : in FILE_TYPE) return BOOLEAN;

 -- Control of default input and output files

 procedure SET_INPUT (FILE : in FILE_TYPE);
 procedure SET_OUTPUT(FILE : in FILE_TYPE);

 function STANDARD_INPUT return FILE_TYPE;
 function STANDARD_OUTPUT return FILE_TYPE;
 function CURRENT_INPUT return FILE_TYPE;
 function CURRENT_OUTPUT return FILE_TYPE;
```

*-- Specification of line and page lengths*

```
procedure SET_LINE_LENGTH(FILE : in FILE_TYPE; TO : in COUNT);
procedure SET_LINE_LENGTH(TO : in COUNT);
```

```
procedure SET_PAGE_LENGTH(FILE : in FILE_TYPE; TO : in COUNT);
procedure SET_PAGE_LENGTH(TO : in COUNT);
```

```
function LINE_LENGTH(FILE : in FILE_TYPE) return COUNT;
function LINE_LENGTH return COUNT;
```

```
function PAGE_LENGTH(FILE : in FILE_TYPE) return COUNT;
function PAGE_LENGTH return COUNT;
```

*-- Column, Line, and Page Control*

```
procedure NEW_LINE (FILE : in FILE_TYPE; SPACING : in POSITIVE_COUNT := 1);
procedure NEW_LINE (SPACING : in POSITIVE_COUNT := 1);
```

```
procedure SKIP_LINE (FILE : in FILE_TYPE; SPACING : in POSITIVE_COUNT := 1);
procedure SKIP_LINE (SPACING : in POSITIVE_COUNT := 1);
```

```
function END_OF_LINE(FILE : in FILE_TYPE) return BOOLEAN;
function END_OF_LINE return BOOLEAN;
```

```
procedure NEW_PAGE (FILE : in FILE_TYPE);
procedure NEW_PAGE;
```

```
procedure SKIP_PAGE (FILE : in FILE_TYPE);
procedure SKIP_PAGE;
```

```
function END_OF_PAGE(FILE : in FILE_TYPE) return BOOLEAN;
function END_OF_PAGE return BOOLEAN;
```

```
function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;
function END_OF_FILE return BOOLEAN;
```

```
procedure SET_COL (FILE : in FILE_TYPE; TO : in POSITIVE_COUNT);
procedure SET_COL (TO : in POSITIVE_COUNT);
```

```
procedure SET_LINE(FILE : in FILE_TYPE; TO : in POSITIVE_COUNT);
procedure SET_LINE(TO : in POSITIVE_COUNT);
```

```
function COL (FILE : in FILE_TYPE) return POSITIVE_COUNT;
function COL return POSITIVE_COUNT;
```

```
function LINE(FILE : in FILE_TYPE) return POSITIVE_COUNT;
function LINE return POSITIVE_COUNT;
```

```
function PAGE(FILE : in FILE_TYPE) return POSITIVE_COUNT;
function PAGE return POSITIVE_COUNT;
```

*-- Character Input-Output*

```

procedure GET(FILE : in FILE_TYPE; ITEM : out CHARACTER);
procedure GET(ITEM : out CHARACTER);
procedure PUT(FILE : in FILE_TYPE; ITEM : in CHARACTER);
procedure PUT(ITEM : in CHARACTER);

-- String Input-Output

procedure GET(FILE : in FILE_TYPE; ITEM : out STRING);
procedure GET(ITEM : out STRING);
procedure PUT(FILE : in FILE_TYPE; ITEM : in STRING);
procedure PUT(ITEM : in STRING);

procedure GET_LINE(FILE : in FILE_TYPE; ITEM : out STRING; LAST : out NATURAL);
procedure GET_LINE(ITEM : out STRING; LAST : out NATURAL);
procedure PUT_LINE(FILE : in FILE_TYPE; ITEM : in STRING);
procedure PUT_LINE(ITEM : in STRING);

-- Generic package for Input-Output of Integer Types

generic
 type NUM is range <>;
package INTEGER_IO is

 DEFAULT_WIDTH : FIELD := NUM'WIDTH;
 DEFAULT_BASE : NUMBER_BASE := 10;

 procedure GET(FILE : in FILE_TYPE; ITEM : out NUM; WIDTH : in FIELD := 0);
 procedure GET(ITEM : out NUM; WIDTH : in FIELD := 0);

 procedure PUT(FILE : in FILE_TYPE;
 ITEM : in NUM;
 WIDTH : in FIELD := DEFAULT_WIDTH;
 BASE : in NUMBER_BASE := DEFAULT_BASE);
 procedure PUT(ITEM : in NUM;
 WIDTH : in FIELD := DEFAULT_WIDTH;
 BASE : in NUMBER_BASE := DEFAULT_BASE);
 procedure GET(FROM : in STRING; ITEM : out NUM; LAST : out POSITIVE);
 procedure PUT(TO : out STRING;
 ITEM : in NUM;
 BASE : in NUMBER_BASE := DEFAULT_BASE);

end INTEGER_IO;

-- Generic packages for Input-Output of Real Types

generic
 type NUM is digits <>;
package FLOAT_IO is

 DEFAULT_FORE : FIELD := 2;
 DEFAULT_AFT : FIELD := NUM'DIGITS-1;
 DEFAULT_EXP : FIELD := 3;

```



```

procedure GET(FILE : in FILE_TYPE; ITEM : out NUM; WIDTH : in FIELD := 0);
procedure GET(ITEM : out NUM; WIDTH : in FIELD := 0);

procedure PUT(FILE : in FILE_TYPE;
 ITEM : in NUM;
 FORE : in FIELD := DEFAULT_FORE;
 AFT : in FIELD := DEFAULT_AFT;
 EXP : in FIELD := DEFAULT_EXP);
procedure PUT(ITEM : in NUM;
 FORE : in FIELD := DEFAULT_FORE;
 AFT : in FIELD := DEFAULT_AFT;
 EXP : in FIELD := DEFAULT_EXP);

procedure GET(FROM : in STRING; ITEM : out NUM; LAST : out POSITIVE);
procedure PUT(TO : out STRING;
 ITEM : in NUM;
 AFT : in FIELD := DEFAULT_AFT;
 EXP : in FIELD := DEFAULT_EXP);
end FLOAT_IO;

generic
 type NUM is delta <>;
package FIXED_IO is

 DEFAULT_FORE : FIELD := NUM'FORE;
 DEFAULT_AFT : FIELD := NUM'AFT;
 DEFAULT_EXP : FIELD := 0;

 procedure GET(FILE : in FILE_TYPE; ITEM : out NUM; WIDTH : in FIELD := 0);
 procedure GET(ITEM : out NUM; WIDTH : in FIELD := 0);

 procedure PUT(FILE : in FILE_TYPE;
 ITEM : in NUM;
 FORE : in FIELD := DEFAULT_FORE;
 AFT : in FIELD := DEFAULT_AFT;
 EXP : in FIELD := DEFAULT_EXP);
 procedure PUT(ITEM : in NUM;
 FORE : in FIELD := DEFAULT_FORE;
 AFT : in FIELD := DEFAULT_AFT;
 EXP : in FIELD := DEFAULT_EXP);

 procedure GET(FROM : in STRING; ITEM : out NUM; LAST : out POSITIVE);
 procedure PUT(TO : out STRING;
 ITEM : in NUM;
 AFT : in FIELD := DEFAULT_AFT;
 EXP : in FIELD := DEFAULT_EXP);
end FIXED_IO;

-- Generic package for Input-Output of Enumeration Types

```

```

generic
 type ENUM is (<>);
package ENUMERATION_IO is

 DEFAULT_WIDTH : FIELD := 0;
 DEFAULT_SETTING : TYPE_SET := UPPER_CASE;

 procedure GET(FILE : in FILE_TYPE; ITEM : out ENUM);
 procedure GET(ITEM : out ENUM);

 procedure PUT(FILE : in FILE_TYPE;
 ITEM : in ENUM;
 WIDTH : in FIELD := DEFAULT_WIDTH;
 SET : in TYPE_SET := DEFAULT_SETTING);
 procedure PUT(ITEM : in ENUM;
 WIDTH : in FIELD := DEFAULT_WIDTH;
 SET : in TYPE_SET := DEFAULT_SETTING);

 procedure GET(FROM : in STRING; ITEM : out ENUM; LAST : out POSITIVE);
 procedure PUT(TO : out STRING;
 ITEM : in ENUM;
 SET : in TYPE_SET := DEFAULT_SETTING);
end ENUMERATION_IO;

-- Exceptions

STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
MODE_ERROR : exception renames IO_EXCEPTIONS.MODE_ERROR;
NAME_ERROR : exception renames IO_EXCEPTIONS.NAME_ERROR;
USE_ERROR : exception renames IO_EXCEPTIONS.USE_ERROR;
DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
END_ERROR : exception renames IO_EXCEPTIONS.END_ERROR;
DATA_ERROR : exception renames IO_EXCEPTIONS.DATA_ERROR;
LAYOUT_ERROR : exception renames IO_EXCEPTIONS.LAYOUT_ERROR;

private
 -- implementation-dependent
end TEXT_IO;

```

## Appendix E— Specification of the Packages Sequential\_IO and Direct\_IO

This appendix contains the specification of the predefined packages Sequential\_IO and Direct\_IO. They are taken directly from the Ada *Language Reference Manual*. In these listings, reserved words are shown in lowercase. All other identifiers are shown in all uppercase.

```
with IO_EXCEPTIONS;
generic
 type ELEMENT_TYPE is private;
package SEQUENTIAL_IO is

 type FILE_TYPE is limited private;

 type FILE_MODE is (IN_FILE, OUT_FILE);

 -- File management

 procedure CREATE(FILE : in out FILE_TYPE;
 MODE : in FILE_MODE := OUT_FILE;
 NAME : in STRING := "";
 FORM : in STRING := "");

 procedure OPEN (FILE : in out FILE_TYPE;
 MODE : in FILE_MODE;
 NAME : in STRING;
 FORM : in STRING := "");

 procedure CLOSE (FILE : in out FILE_TYPE);
 procedure DELETE(FILE : in out FILE_TYPE);
 procedure RESET (FILE : in out FILE_TYPE; MODE : in FILE_MODE);
 procedure RESET (FILE : in out FILE_TYPE);

 function MODE (FILE : in FILE_TYPE) return FILE_MODE;
 function NAME (FILE : in FILE_TYPE) return STRING;
 function FORM (FILE : in FILE_TYPE) return STRING;

 function IS_OPEN(FILE : in FILE_TYPE) return BOOLEAN;

 -- Input and output operations

 procedure READ (FILE : in FILE_TYPE; ITEM : out ELEMENT_TYPE);
 procedure WRITE (FILE : in FILE_TYPE; ITEM : in ELEMENT_TYPE);

 function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;

 -- Exceptions

 STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
 MODE_ERROR : exception renames IO_EXCEPTIONS.MODE_ERROR;
 NAME_ERROR : exception renames IO_EXCEPTIONS.NAME_ERROR;
 USE_ERROR : exception renames IO_EXCEPTIONS.USE_ERROR;
 DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
 END_ERROR : exception renames IO_EXCEPTIONS.END_ERROR;
 DATA_ERROR : exception renames IO_EXCEPTIONS.DATA_ERROR;
```

```

private
 -- implementation-dependent
end SEQUENTIAL_IO;

with IO_EXCEPTIONS;
generic
 type ELEMENT_TYPE is private;
package DIRECT_IO is

 type FILE_TYPE is limited private;

 type FILE_MODE is (IN_FILE, INOUT_FILE, OUT_FILE);
 type COUNT is range 0 .. implementation defined;
 subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;

 -- File management

 procedure CREATE(FILE : in out FILE_TYPE;
 MODE : in FILE_MODE := INOUT_FILE;
 NAME : in STRING := "";
 FORM : in STRING := "");

 procedure OPEN (FILE : in out FILE_TYPE;
 MODE : in FILE_MODE;
 NAME : in STRING;
 FORM : in STRING := "");

 procedure CLOSE (FILE : in out FILE_TYPE);
 procedure DELETE(FILE : in out FILE_TYPE);
 procedure RESET (FILE : in out FILE_TYPE; MODE : in FILE_MODE);
 procedure RESET (FILE : in out FILE_TYPE);

 function MODE (FILE : in FILE_TYPE) return FILE_MODE;
 function NAME (FILE : in FILE_TYPE) return STRING;
 function FORM (FILE : in FILE_TYPE) return STRING;

 function IS_OPEN(FILE : in FILE_TYPE) return BOOLEAN;

 -- Input and output operations

 procedure READ (FILE : in FILE_TYPE; ITEM : out ELEMENT_TYPE; FROM : POSITIVE_COUNT);
 procedure READ (FILE : in FILE_TYPE; ITEM : out ELEMENT_TYPE);

 procedure WRITE(FILE : in FILE_TYPE; ITEM : in ELEMENT_TYPE; TO : POSITIVE_COUNT);
 procedure WRITE(FILE : in FILE_TYPE; ITEM : in ELEMENT_TYPE);

 procedure SET_INDEX(FILE : in FILE_TYPE; TO : in POSITIVE_COUNT);

 function INDEX(FILE : in FILE_TYPE) return POSITIVE_COUNT;
 function SIZE (FILE : in FILE_TYPE) return COUNT;

 function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;

```

*-- Exceptions*

```
STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
MODE_ERROR : exception renames IO_EXCEPTIONS.MODE_ERROR;
NAME_ERROR : exception renames IO_EXCEPTIONS.NAME_ERROR;
USE_ERROR : exception renames IO_EXCEPTIONS.USE_ERROR;
DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
END_ERROR : exception renames IO_EXCEPTIONS.END_ERROR;
DATA_ERROR : exception renames IO_EXCEPTIONS.DATA_ERROR;
```

```
private
```

```
 -- implementation-dependent
```

```
end DIRECT_IO;
```

## Appendix F— Specification of the Package Calendar

This appendix contains the specification of the predefined package Calendar. It is taken directly from the Ada *Language Reference Manual*. In this listing, reserved words are lowercase. All other identifiers are uppercase.

```
package CALENDAR is
 type TIME is private;

 subtype YEAR_NUMBER is INTEGER range 1901 .. 2099;
 subtype MONTH_NUMBER is INTEGER range 1 .. 12;
 subtype DAY_NUMBER is INTEGER range 1 .. 31;
 subtype DAY_DURATION is DURATION range 0.0 .. 86_400.0;

 function CLOCK return TIME;

 function YEAR (DATE : TIME) return YEAR_NUMBER;
 function MONTH (DATE : TIME) return MONTH_NUMBER;
 function DAY (DATE : TIME) return DAY_NUMBER;
 function SECONDS (DATE : TIME) return DAY_DURATION;

 procedure SPLIT (DATE : in TIME;
 YEAR : out YEAR_NUMBER;
 MONTH : out MONTH_NUMBER;
 DAY : out DAY_NUMBER;
 SECONDS : out DAY_DURATION);

 function TIME_OF (YEAR : YEAR_NUMBER;
 MONTH : MONTH_NUMBER;
 DAY : DAY_NUMBER;
 SECONDS : DAY_DURATION := 0.0) return TIME;

 function "+" (LEFT : TIME; RIGHT : DURATION) return TIME;
 function "+" (LEFT : DURATION; RIGHT : TIME) return TIME;
 function "-" (LEFT : TIME; RIGHT : DURATION) return TIME;
 function "-" (LEFT : TIME; RIGHT : TIME) return DURATION;

 function "<" (LEFT, RIGHT : TIME) return BOOLEAN;
 function "<=" (LEFT, RIGHT : TIME) return BOOLEAN;
 function ">" (LEFT, RIGHT : TIME) return BOOLEAN;
 function ">=" (LEFT, RIGHT : TIME) return BOOLEAN;

 TIME_ERROR : exception; -- can be raised by TIME_OF, "+", and "-"

private
 -- implementation-dependent
end;
```

## Appendix G— Specification of the Package Standard

This appendix contains the specification of the predefined package Standard. It is taken directly from the Ada *Language Reference Manual*. In this listing, reserved words are lowercase. All other identifiers are uppercase. The operators that are predefined for the types declared in the package Standard are given in comments because they are implicitly declared. Italics are used for pseudonames of anonymous types (such as *universal real*) and for information (such as *implementation\_defined* and *any\_fixed\_point\_type*).

```
package STANDARD is

 type BOOLEAN is (FALSE, TRUE);

 -- The predefined relational operators for this type are as follows:

 -- function "=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
 -- function "/=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
 -- function "<" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
 -- function "<=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
 -- function ">" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
 -- function ">=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;

 -- The predefined logical operators and the predefined logical negation operator
 -- are as follows:

 -- function "and" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
 -- function "or" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
 -- function "xor" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;

 -- function "not" (RIGHT : BOOLEAN) return BOOLEAN;

 -- The universal type universal_integer is predefined.

 type INTEGER is implementation_defined;

 -- The predefined operators for this type are as follows:

 -- function "=" (LEFT, RIGHT : INTEGER) return BOOLEAN;
 -- function "/=" (LEFT, RIGHT : INTEGER) return BOOLEAN;
 -- function "<" (LEFT, RIGHT : INTEGER) return BOOLEAN;
 -- function "<=" (LEFT, RIGHT : INTEGER) return BOOLEAN;
 -- function ">" (LEFT, RIGHT : INTEGER) return BOOLEAN;
 -- function ">=" (LEFT, RIGHT : INTEGER) return BOOLEAN;

 -- function "+" (RIGHT : INTEGER) return INTEGER;
 -- function "-" (RIGHT : INTEGER) return INTEGER;
 -- function "abs" (RIGHT : INTEGER) return INTEGER;

 -- function "+" (LEFT, RIGHT : INTEGER) return INTEGER;
 -- function "-" (LEFT, RIGHT : INTEGER) return INTEGER;
 -- function "*" (LEFT, RIGHT : INTEGER) return INTEGER;
```

```

-- function "/" (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "rem" (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "mod" (LEFT, RIGHT : INTEGER) return INTEGER;

-- function "***" (LEFT : INTEGER; RIGHT : INTEGER) return INTEGER;

-- An implementation may provide additional predefined integer types.
-- It is recommended that the names of such additional types end with
-- INTEGER as in SHORT_INTEGER or LONG_INTEGER. The specification of
-- each operator for the type universal_integer, or for any additional
-- predefined integer type, is obtained by replacing INTEGER by
-- the name of the type in the specification of the corresponding
-- operator of the type INTEGER, except for the right operand of the
-- exponentiating operator.

-- The universal type universal_real is predefined.

type FLOAT is implementation_defined;

-- The predefined operators for this type are as follows:

-- function "=" (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function "<" (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function ">" (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : FLOAT) return BOOLEAN;

-- function "+" (RIGHT : FLOAT) return FLOAT;
-- function "-" (RIGHT : FLOAT) return FLOAT;
-- function "abs" (RIGHT : FLOAT) return FLOAT;

-- function "+" (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "-" (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "*" (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "/" (LEFT, RIGHT : FLOAT) return FLOAT;

-- function "***" (LEFT : FLOAT; RIGHT : INTEGER) return FLOAT;

-- An implementation may provide additional predefined floating point
-- types. It is recommended that the names of such additional
-- types end with FLOAT as in SHORT_FLOAT or LONG_FLOAT. The
-- specification of each operator for the type universal_real, or for
-- any additional predefined floating point type, is obtained by
-- replacing FLOAT by the name of the type in the specification of the
-- corresponding operator of the type FLOAT.

-- In addition, the following operators are predefined for universal types:

-- function "*" (LEFT : universal_integer; RIGHT : universal_real)
-- return universal_real;

```



```

-- function "*" (LEFT : universal_real; RIGHT : universal_integer)
-- return universal_real;

-- function "/" (LEFT : universal_real; RIGHT : universal_integer)
-- return universal_real;

-- The type universal_fixed is predefined.
-- The only operators declared for this type are

-- function "*" (LEFT : any_fixed_point_type; RIGHT : any_fixed_point_type)
-- return universal_fixed;

-- function "/" (LEFT : any_fixed_point_type; RIGHT : any_fixed_point_type)
-- return universal_fixed;

-- The following characters form the standard ASCII character set.
-- Character literals -- corresponding to control characters are not
-- identifiers; they are indicated in italics in this definition.

type CHARACTER is
 (nul, soh, stx, etx, eot, enq, ack, bel,
 bs, ht, lf, vt, ff, cr, so, si,
 dle, dcl, dc2, dc3, dc4, nak, syn, etb,
 can, em, sub, esc, fs, gs, rs, us,

 ' ', '!', '"', '#', '$', '%', '&', ' ' ',
 '(', ')', '*', '+', ',', '-', '.', '/',
 '0', '1', '2', '3', '4', '5', '6', '7',
 '8', '9', ':', ';', '<', '=', '>', '?',

 '@', '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', '[', '\', ']', '~', '_',

 '`', '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', '{', '|', '}', '~', del);

for CHARACTER use -- 128 ASCII character set without holes
 (0, 1, 2, 3, 4, 5, . . ., 125, 126, 127);

-- The predefined operators for the type CHARACTER are the same as
-- for any enumeration type.

```

```
package ASCII is
```

```
-- Control characters:
```

```
NUL : constant CHARACTER := nul; SOH : constant CHARACTER := soh;
STX : constant CHARACTER := stx; ETX : constant CHARACTER := etx;
EOT : constant CHARACTER := eot; ENQ : constant CHARACTER := enq;
ACK : constant CHARACTER := ack; BEL : constant CHARACTER := bel;
BS : constant CHARACTER := bs; HT : constant CHARACTER := ht;
LF : constant CHARACTER := lf; VT : constant CHARACTER := vt;
FF : constant CHARACTER := ff; CR : constant CHARACTER := cr;
SO : constant CHARACTER := so; SI : constant CHARACTER := si;
DLE : constant CHARACTER := dle; DC1 : constant CHARACTER := dc1;
DC2 : constant CHARACTER := dc2; DC3 : constant CHARACTER := dc3;
DC4 : constant CHARACTER := dc4; NAK : constant CHARACTER := nak;
SYN : constant CHARACTER := syn; ETB : constant CHARACTER := etb;
CAN : constant CHARACTER := can; EM : constant CHARACTER := em;
SUB : constant CHARACTER := sub; ESC : constant CHARACTER := esc;
FS : constant CHARACTER := fs; GS : constant CHARACTER := gs;
RS : constant CHARACTER := rs; US : constant CHARACTER := us;
DEL : constant CHARACTER := del;
```

```
-- Other characters:
```

```
EXCLAM : constant CHARACTER := '!'; QUOTATION : constant CHARACTER := '"';
SHARP : constant CHARACTER := '#'; DOLLAR : constant CHARACTER := '$';
PERCENT : constant CHARACTER := '%'; AMPERSAND : constant CHARACTER := '&';
COLON : constant CHARACTER := ':'; SEMICOLON : constant CHARACTER := ';';
QUERY : constant CHARACTER := '?'; AT_SIGN : constant CHARACTER := '@';
L_BRACKET : constant CHARACTER := '['; BACK_SLASH : constant CHARACTER := '\';
R_BRACKET : constant CHARACTER := ']'; CIRCUMFLEX : constant CHARACTER := '~';
UNDERLINE : constant CHARACTER := '_'; GRAVE : constant CHARACTER := '`';
L_BRACE : constant CHARACTER := '{'; BAR : constant CHARACTER := '|';
R_BRACE : constant CHARACTER := '}'; TILDE : constant CHARACTER := '~';
```

```
-- Lower case letters:
```

```
LC_A : constant CHARACTER := 'a';
. . .
LC_Z : constant CHARACTER := 'z';
```

```
end ASCII;
```

```
-- Predefined subtypes:
```

```
subtype NATURAL is INTEGER range 0 .. INTEGER'LAST;
subtype POSITIVE is INTEGER range 1 .. INTEGER'LAST;
```

```

-- Predefined string type:

type STRING is array (POSITIVE range <>) of CHARACTER;

pragma PACK(STRING);

-- The predefined operators for this type are as follows:

-- function "=" (LEFT, RIGHT : STRING) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : STRING) return BOOLEAN;
-- function "<" (LEFT, RIGHT : STRING) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : STRING) return BOOLEAN;
-- function ">" (LEFT, RIGHT : STRING) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : STRING) return BOOLEAN;

-- function "&" (LEFT : STRING; RIGHT : STRING) return STRING;
-- function "&" (LEFT : CHARACTER; RIGHT : STRING) return STRING;
-- function "&" (LEFT : STRING; RIGHT : CHARACTER) return STRING;
-- function "&" (LEFT : CHARACTER; RIGHT : CHARACTER) return STRING;

type DURATION is delta implementation_defined range implementation_defined;

-- The predefined operators for the type DURATION are the same as for
-- any fixed point type.

-- The predefined exceptions:

CONSTRAINT_ERROR : exception;
NUMERIC_ERROR : exception;
PROGRAM_ERROR : exception;
STORAGE_ERROR : exception;
TASKING_ERROR : exception;

end STANDARD;
```

## Appendix H— Program Style, Formatting, and Documentation

Throughout this text we encourage the use of good programming style and documentation. Although the programs you write for class assignments may not be looked at by anyone except the person grading your work, outside of class you will write programs that will be used by others.

Useful programs have very long lifetimes, during which they must be modified and updated. When maintenance work must be done, either you or another programmer will have to do it. Good style and documentation are essential if another programmer is to understand and work with your program. You also will discover that, after not working with your own program for a few months, you'll forget many of the details.

### *General Guidelines*

The style used in the programs and fragments throughout this text provides a good starting point for developing your own style. Our goals in creating this style were to make it simple, consistent, and easy to read.

Style is of benefit only for a human reader of your program—differences in style make no difference to the computer. Good style includes the use of meaningful identifiers, comments, and indentation of control structures, all of which help others to understand and work with your program. Perhaps the most important aspect of program style is consistency. If the style within a program is not consistent, then it becomes misleading and confusing.

Sometimes, your instructor or the company you work for will specify a particular style. When you are modifying someone else's code, you will use his or her style in order to maintain consistency within the program. However, you also will develop your own personal programming style based on what you've been taught, your experience, and your personal taste.

### *Comments*

Comments are extra information included to make a program easier to understand. You should include a comment wherever the code is difficult to understand. However, don't overcomment. Too many comments in a program will obscure the code and be a source of distraction.

In our style, there are four basic types of comments: headers, definitions, inline, and sidebar.

1. *Header comments* appear at the top of the program immediately following the specification and should include your name, the date that the program was written, and its purpose. It is also useful to include the input, output, and assumptions sections from your top-down design. Think of the header comments as the reader's introduction to your program. Here is an example.

```
--
-- This program computes the sidereal time for a given date and solar time.
--
-- Written By: Your Name
--
-- Date Completed: 4/8/94
--
-- Input: A date and time in the form of MM DD YY HH MM SS
--
--
--
-- Output: Sidereal time in the form of HH MM SS
--
--
-- Assumptions: Solar time is specified for a longitude of 0
-- degrees (GMT, UT, or Z time zone)
```

You should also include header comments for all programmer-defined procedures, functions, and packages.

2. *Definition comments* accompany all declarations and definitions in the program. Wherever an identifier is declared, you should include a comment that explains its purpose. In programs in the text, definition comments appear to the right of the identifier being declared. For example

```
E : constant := 2.718281828459 -- The base of the natural logarithms

Delta_X : Float; -- The difference in the X direction
Delta_Y : Float; -- The difference in the Y direction
```

Notice that aligning the comments gives the code a neater appearance and is less distracting.

3. *Inline comments* are used to break long sections of code into shorter, more comprehensible fragments. These are usually the names of modules in your top-down design, although you may occasionally choose to include other information. You should surround inline comments with blank lines to make them stand out. (We have not always done this in this text. To save space, we have used italics instead.) For example

```
begin -- Main Program

 -- Initialize

 Text_IO.Open (File => In_File,
 Mode => Text_IO.In_File,
 Name => "TEMP.DAT");
 Min_Temp := System.Max_Int;

 -- Get Data

 Text_IO.Get (File => In_File, Item => Num_Temps);
```

You may insert blank lines wherever there is a logical break in the code that you would like to emphasize.

4. *Sidebar comments* appear to the right of statements in the body of the program and are used to shed light on the function of the statement. Sidebar comments are often pseudocode statements from the lowest levels of your top-down design. If a complicated Ada statement requires some explanation, you should write the pseudocode statement to the right of the Ada statement. For example

```
if Text_IO.End_Of_File (File1) /= -- If one of the files is empty
 Text_IO.End_Of_File (File2) then
 .
 .
 .
```

In addition to the four main types of comments that we have discussed, there are two other miscellaneous uses for comments that we should mention: (1) You should include comments at the beginning of each procedure, function, and package; (2) A row of dashes should appear before and after each procedure or function to help it to stand out. For example

```

procedure Balance (Param_1 : out Float) is -- Definition of Param_1

begin -- Balance
 .
 .
end Balance;

```

### ***Identifiers and Reserved Words***

The most important consideration in choosing a name for an object or process in a program is that the name convey as much information as possible about what the object is or what the process does. The name should also be readable in the context in which it is used. For example, the following names convey the same information, but one is more readable than the other:

Date\_Of\_Inv Invoice\_Date

In addition to picking an informative name for each identifier, you should choose names for different identifiers that are *psychologically distant*\* enough to avoid confusion. Words that look alike or have similar meanings are not psychologically distant. The following table ranks some example identifier name pairs in terms of their psychological distance.

---

\*The application of psychological principles to programming is discussed by Gerald M. Weinberg in his book. *The Psychology of Computer Programming*. New York: John Wiley & Sons, Inc., 1971.

| <i>First Identifier</i> | <i>Second Identifier</i> | <i>Psychological Distance</i> |
|-------------------------|--------------------------|-------------------------------|
| Sum_1                   | Sum_2                    | Very little (look alike)      |
| Sum                     | Total                    | Little (same meaning)         |
| Male_Total              | Female_Total             | Big (very informative)        |

Use nouns as identifiers for types, constants, or variables and verbs as names of procedures. Because of the way that functions are called, use nouns or occasionally adjectives as function names. To distinguish them from variables and constants, it is usually best to end programmer-defined type identifiers with the word *Type*, *Rec*, or *Array*. In some cases, such as the name of a unit of measurement, it is clearer when the word type is not added to the end of the type identifier. Here are some examples.

|                   |                                                                         |
|-------------------|-------------------------------------------------------------------------|
| <i>Variables</i>  | Name, Car_Make, Price, Hours, Month, Paper_Length                       |
| <i>Constants</i>  | Pi, Tax_Rate, String_Length, Array_Size                                 |
| <i>Data Types</i> | Name_Type, Car_Make_Type, Month_Type, Inches, Customer_Rec, Score_Array |
| <i>Procedures</i> | Get_Data, Clear_Table, Print_Bar_Chart                                  |
| <i>Functions</i>  | Cube_Root, Greatest, Color, Area_Of, To_Inches                          |

Although an identifier may be a series of words, very long identifiers can become quite tedious, making the program difficult to read. One way to shorten long identifiers is to use acronyms or abbreviations. Use capital letters for acronyms (DNA\_Weight, for example). An abbreviation should be significantly shorter than the word it abbreviates. Try to use only standard abbreviations such as those listed in the back of most dictionaries. Be consistent with your abbreviations. Don't, for example, use both Num and Nmbr as abbreviations of Number in the same program.

Capitalization is another consideration when choosing an identifier. In this text's programs, the first letter of every word in an identifier is capitalized, and Ada reserved words are lowercase letters to distinguish them from identifiers.

The best approach to designing an identifier is to try writing out different names until you reach an acceptable compromise, and then write an especially informative definition comment next to the declaration.

### ***Declaration Order***

Declarations for named numbers, constants, variables, subprograms, types, and subtypes are made in the declarative part of our program or subprogram. The instantiation of generic packages also is done in this part of our program. Ada allows flexibility in the order of declarations. To maintain a consistent style, we generally make our declarations in the following order.

1. Declaration of named numbers
2. Declaration of programmer-defined types and subtypes
3. Declaration of named constants of programmer-defined types and subtypes
4. Instantiations of generic packages
5. Declaration of variables
6. Declaration of procedures and functions

However, because we must define an identifier before we can use it, there are situations in which we cannot group all similar declarations together in this order. In such cases, we modify the order of declarations or mix different kinds of declarations. There is no particular advantage of this ordering over others. Our ordering is similar to the order required by some other languages. Consistency is the key to readable programs.

We precede each of these declaration categories by an inline comment. We indent them all in the same way. We list each identifier, one per line, and indent it by three spaces from the level of the procedure, function, or package specification with a defining comment to the right. Here is an example.

```

procedure Declaration_Example is

 -- Named Numbers

 Pi : constant := 3.141592654; -- Ratio of circumference to diameter
 E : constant := 2.718281828; -- Base of the natural logarithms

 -- Types

 type Product_Type is (Lumber, Plywood); -- Products offered
 type Hours_Type is range 0..100; -- Hours worked

 subtype Name_Type is String(1..30); -- For names

 type Time_Rec is
 record -- Time clock data
 First : Name_Type; -- Employee first name
 Last : Name_Type; -- Employee last name
 Hours : Hours_Type; -- Hours worked this week
 end record;

 -- Named Constants

 Max_Hours :
 constant Hours_Type := 40; -- Hours worked before overtime
 -- must be paid

 -- Instantiated Packages

 package Product_IO is new Text_IO Enumeration_IO (Enum => Product_Type);
 package Integer_IO is new Text_IO.Integer_IO (Num => Integer);
 package Float_IO is new Text_IO.Float_IO (Num => Float);

 -- Variables

 Radius : Float; -- Radius of a circle
 Diameter : Float; -- Diameter of a circle
 Area : Float; -- Area of a circle
 Count : Integer; -- Example number
 Max_Circle : Integer; -- Number of greatest circle
 Min_Circle : Integer; -- Number of smallest circle
 .
 .
 .

```



In the text, we use two different indentation styles in procedure and function declarations. When space permits, the parameter list begins on the same line as the procedure or function name. Each formal parameter is on a separate line, in a style similar to that used for variable declarations as shown in the following example.

```
procedure Get_Sum (Number : in Integer; -- Number of values to be read
 Sum : out Float) is -- Sum of values read
```

Occasionally, we need more room for the comments. In such a case, we use a second style in which the parameter list appears on lines following the procedure or function name. In the following example, note that the parameter list is indented under the procedure name specification.

```
procedure Get_Data_Average
 (In_file : in out Text_IO.File_Type; -- File of test scores
 Num_Scores : in Integer; -- Number of scores to read
 Average: : out Float); is -- Average of the test scores
```

The local declarations and body of a procedure or function follow the same guidelines as the declarations and body of the main program. The only difference is that nested procedures or functions should be indented by three spaces for each level of nesting.

### ***Formatting Lines and Expressions***

In general, it is best never to include more than one statement on a line. When you must break a long statement in the middle and continue it on the next line, it's important to choose a breaking point that is logical and readable. Compare the readability of the following code fragments.

```
Text_IO.Put ("When you use strings that are too long to fit on a sin"&
 "gle line, break them at natural boundaries.");
```

```
Text_IO.Put ("When you use strings that are too long to fit on a " &
 "single line, break them at natural boundaries.");
```

When you must split an expression across multiple lines, try to end each line with an operator. Also, try to take advantage of any repeating patterns in the expression. For example

```
Mean_Of_Maxima := (Maximum(Set1_Value1, Set1_Value2, Set1_Value3) +
 Maximum(Set2_Value1, Set2_Value2, Set2_Value3) +
 Maximum(Set3_Value1, Set3_Value2, Set3_Value3)) / 3.0;
```

```
Total_Amount := 0.0; -- These three assignment statements are
Num := 0; -- aligned at the assignment operator (:=)
Male_Count := 0; -- for easier reading
```

When writing expressions, also keep in mind that spaces improve readability. Usually you should include one space on either side of the := and most operators. Occasionally

spaces are left out to emphasize the order in which operations are performed. Here are some examples.

```
if X+Y > Y+Z then
 Maximum := X + Y
else
 Maximum := Y + Z;
```

```
Poly_Result := 3.8*X**3 - 4.5*X**2 - 2.3*X + 7.2
```

### ***Indentation***

The purpose of indenting statements in a program is to provide visual cues to the reader and to make the program easier to debug. When a program is properly indented, the way the statements are grouped is immediately obvious. Compare the following two program fragments.

|                                                                                                                                                                                                                                                                   |                                                                                                                                                                                                                                                                                                                                                   |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>Count_Loop: loop exit Count_Loop when Count &gt; 10 Integer_IO.Get (Num); if Num = 0 then Count := Count + 1; Num := 1; end if; Integer_IO.Put (Item =&gt; Num, Width =&gt; 1); Integer_IO.Put (Item =&gt; Count, Width =&gt; 1); end loop Count_Loop;</pre> | <pre>Count_loop: loop     exit Count_Loop when Count &gt; 10     Integer_IO.Get (Num);     if Num = 0 then         Count := Count + 1;         Num := 1;     end if;     Integer_IO.Put (Item =&gt; Num,                     Width =&gt; 1);     Integer_IO.Put (Item =&gt; Count,                     Width =&gt; 1); end loop Count_Loop;</pre> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

As a basic rule in this text, we have indented each nested or lower-level item by three spaces. Exceptions to this rule are formal parameters and statements that are split across two or more lines. Indenting by three spaces is really just a minimum. Many people prefer to indent by four or even five spaces.

### **Statements**

In general, you should indent any statement that is part of another statement, including assignment statements, procedure calls, and nested statements. The following example Ada statements follow these indentation guidelines.

```
if Sex = Male then
 Male_Salary := Male_Salary + Salary;
 Male_Count := Male_Count + 1;
else
 Female_Salary := Female_Salary + Salary;
end if;

if Count > 0 then
 Average := Total / Count;
end if;
```

```

if Month = "Jan" then
 Month_Num := 1;
elsif Month = "Feb" then
 Month_Num := 2;
elsif Month = "Mar" then
 Month_Num := 3;
.
.
.
elsif Month = "Nov" then
 Month_Num := 11;
else
 Month_Num := 12;
end if;

case Color is
 when Red | Blue | Green =>
 Color_IO.Put (Color);
 Text_IO.Put (" is an additive primary.");
 when Cyan | Magenta | Yellow =>
 Color_IO.Put (Color);
 Text_IO.Put (" is a subtractive primary." ;
 when White | Black =>
 Text_IO.Put ("Not valid color.");
 when others =>
 Text_IO.Put ("Not a primary color.");
end case;

Basic_Loop:
loop
 exit Basic_Loop when Count >= 10;
 Integer_IO.Get (Value);
 Total := Total - Value;
 Count := Count + 1;
end loop Basic_Loop;

While_Loop:
while Count < 10 loop
 Integer_IO.Get (Value);
 Total := Total + Value;
 Count := Count + 1;
end loop While_Loop;

For_Loop:
for Count in 1..9 loop
 Integer_IO.Get (Value);
 Total := Total + Value;
end loop For_Loop;

```

## Appendix I— Predefined Language Attributes

This appendix summarizes the attributes predefined by the Ada language.\*

| <i>Attribute</i> | <i>Definition</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| P'ADDRESS        | <p>For a prefix P that denotes an object, a program unit, a level, or an entry:</p> <p>Yields the address of the first of the storage units allocated to P. For a subprogram, package, task unit, or label, this value refers to the machine code associated with the corresponding body or statement. For an entry for which an address clause has been given, the value refers to the corresponding hardware interrupt. The value of this attribute is of the type ADDRESS defined in the package SYSTEM.</p> |
| P'AFT            | <p>For a prefix P that denotes a fixed-point subtype:</p> <p>Yields the number of decimal digits needed after the point to accommodate the precision of the subtype P, unless the delta of the subtype P is greater than 0.1, in which case the attribute yields the value one. (P'AFT is the smallest positive integer N for which <math>(100*N)*P'DELTA</math> is greater than or equal to one.) The value of this attribute is of the type universal_integer.</p>                                            |
| P'BASE           | <p>For a prefix P that denotes a type or subtype:</p> <p>This attribute denotes the base type of P. It is only allowed as the prefix of the name of another attribute: for example, P'BASE'FIRST.</p>                                                                                                                                                                                                                                                                                                           |
| P'CALLABLE       | <p>For a prefix P that is appropriate for a task type:</p> <p>Yields the value FALSE when the execution of the task P is either completed or terminated, or when the task is abnormal; yields the value TRUE otherwise. The value of this attribute is of the predefined type BOOLEAN.</p>                                                                                                                                                                                                                      |
| P'CONSTRAINED    | <p>For a prefix P that denotes an object of a type with discriminants:</p> <p>Yields the value TRUE if a discriminant constraint applies to the object P, or if the object is a constant (including a formal parameter or generic formal parameter of mode in); yields the value FALSE otherwise. If P is a generic formal parameter of</p>                                                                                                                                                                     |

---

\* This material is reproduced with permission from American National Standard *Reference Manual for the Ada Programming Language*, copyright 1983 by the American National Standards Institute. Copies of this standard may be purchased from the American National Standards Institute at 11 West 42nd Street, New York, NY 10036.

| <i>Attribute</i> | <i>Definition</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                  | <p>mode in out, or if P is a formal parameter of mode in out or out and the type mark given in the corresponding parameter specification denotes an unconstrained type with discriminants, then the value of this attribute is obtained from that of the corresponding actual parameter. The value of this attribute is of the predefined type BOOLEAN.</p>                                                                                                      |
| P'CONSTRAINED    | <p>For a prefix P that denotes a private type or subtype:</p> <p>Yields the value FALSE if P denotes an unconstrained nonformal private type with discriminants; also yields the value FALSE if P denotes a generic formal private type and the associated actual subtype is either an unconstrained type with discriminants or an unconstrained array type; yields the value TRUE otherwise. The value of this attribute is of the predefined type BOOLEAN.</p> |
| P'COUNT          | <p>For a prefix P that denotes an entry of a task unit:</p> <p>Yields the number of entry calls presently queued on the entry (if the attribute is evaluated within an accept statement for the entry P, the count does not include the calling task). The value of this attribute is of the type <code>universal_integer</code>.</p>                                                                                                                            |
| P'DELTA          | <p>For a prefix P that denotes a fixed-point subtype:</p> <p>Yields the value of the delta specified in the fixed accuracy definition for the subtype P. The value of this attribute is of the type <code>universal_real</code>.</p>                                                                                                                                                                                                                             |
| P'DIGITS         | <p>For a prefix P that denotes a floating-point subtype:</p> <p>Yields the number of decimal digits in the decimal mantissa of model numbers of the subtype P. The value of this attribute is of the type <code>universal_integer</code>.</p>                                                                                                                                                                                                                    |
| P'EMAX           | <p>For a prefix P that denotes a floating-point subtype:</p> <p>Yields the largest exponent value in the binary canonical form of model numbers of the subtype P. The value of this attribute is of the type <code>universal_integer</code>.</p>                                                                                                                                                                                                                 |
| P'EPSILON        | <p>For a prefix P that denotes a floating-point subtype:</p> <p>Yields the absolute value of the difference between the model number 1.0 and the next model number above, for the subtype P. The value of this attribute is of the type <code>universal_real</code>.</p>                                                                                                                                                                                         |
| P'FIRST          | <p>For a prefix P that denotes a scalar type, or a subtype of a scalar type:</p> <p>Yields the lower bound of P. The value of this attribute has the same type as P.</p>                                                                                                                                                                                                                                                                                         |

| <i>Attribute</i> | <i>Definition</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| P'FIRST          | <p>For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:</p> <p>Yields the lower bound of the first index range. The value of this attribute has the same type as this lower bound.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| P'FIRST(N)       | <p>For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:</p> <p>Yields the lower bound of the N-th index range. The value of this attribute has the same type as this lower bound. The argument N must be a static expression of type <code>universal_integer</code>. The value of N must be positive (nonzero) and no greater than the dimensionality of the array.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| P'FIRST_BIT      | <p>For a prefix P that denotes a component of a record object:</p> <p>Yields the offset, from the start of the first of the storage units occupied by the component, of the first bit occupied by the component. This offset is measured in bits. The value of this attribute is of the type <code>universal_integer</code>.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| P'FORE           | <p>For a prefix P that denotes a fixed-point subtype:</p> <p>Yields the minimum number of characters needed for the integer part of the decimal representation of any value of the subtype P, assuming that the representation does not include an exponent, but includes a one-character prefix that is either a minus sign or a space. (This minimum number does not include superfluous zeros or underlines, and is at least two.) The value of this attribute is of the type <code>universal_integer</code>.</p>                                                                                                                                                                                                                                                                                                                                                                        |
| P'IMAGE          | <p>For a prefix P that denotes a discrete type or subtype:</p> <p>This attribute is a function with a single parameter. The actual parameter X must be a value of the base type of P. The result type is the predefined type <code>STRING</code>. The result is the image of the value of X, that is, a sequence of characters representing the value in display form. The image of an integer value is the corresponding decimal literal; without underlines, leading zeros, exponent, or trailing spaces; but with a one-character prefix that is either a minus sign or a space. The image of an enumeration value is either the corresponding identifier in upper case or the corresponding character literal (including the two apostrophes); neither leading nor trailing spaces are included. The image of a character other than a graphic character is implementation-defined.</p> |

| <i>Attribute</i>            | <i>Definition</i>                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-----------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| P <small>LARGE</small>      | <p>For a prefix P that denotes a real subtype:</p> <p>The attribute yields the largest positive model number of the subtype P. The value of this attribute is of the type <code>universal_real</code>.</p>                                                                                                                                                                                                                                                 |
| P <small>LAST</small>       | <p>For a prefix P that denotes a scalar type, or a subtype of a scalar type:</p> <p>Yields the upper bound of P. The value of this attribute has the same type as P.</p>                                                                                                                                                                                                                                                                                   |
| P <small>LAST</small>       | <p>For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:</p> <p>Yields the upper bound of the first index range. The value of this attribute has the same type as this upper bound.</p>                                                                                                                                                                                                                       |
| P <small>LAST</small> (N)   | <p>For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:</p> <p>Yields the upper bound of the N-th index range. The value of this attribute has the same type as this upper bound. The argument N must be a static expression of type <code>universal_integer</code>. The value of N must be positive (nonzero) and no greater than the dimensionality of the array.</p>                                      |
| P <small>LAST_BIT</small>   | <p>For a prefix P that denotes a component of a record object:</p> <p>Yields the offset, from the start of the first of the storage units occupied by the component, of the last bit occupied by the component. This offset is measured in bits. The value of this attribute is of the type <code>universal_integer</code>.</p>                                                                                                                            |
| P <small>LENGTH</small>     | <p>For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:</p> <p>Yields the number of values of the first index range (zero for a null range). The value of this attribute is of the type <code>universal_integer</code>.</p>                                                                                                                                                                                  |
| P <small>LENGTH</small> (N) | <p>For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:</p> <p>Yields the number of values of the N-th index range (zero for a null range). The value of this attribute is of the type <code>universal_integer</code>. The argument N must be a static expression of type <code>universal_integer</code>. The value of N must be positive (nonzero) and no greater than the dimensionality of the array.</p> |

| <i>Attribute</i>   | <i>Definition</i>                                                                                                                                                                                                                                                                                                                                                                                          |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| P'MACHINE_EMAX     | <p>For a prefix P that denotes a floating-point type or subtype:</p> <p>Yields the largest value of exponent for the machine representation of the base type of P. The value of this attribute is of the type <code>universal_integer</code>.</p>                                                                                                                                                          |
| P'MACHINE_EMIN     | <p>For a prefix P that denotes a floating-point type or subtype:</p> <p>Yields the smallest (most negative) value of exponent for the machine representation of the base type of P. The value of this attribute is of the type <code>universal_integer</code>.</p>                                                                                                                                         |
| P'MACHINE_MANTISSA | <p>For a prefix P that denotes a floating-point type or subtype:</p> <p>Yields the number of digits in the mantissa for the machine representation of the base type of P (the digits are extended digits in the range 0 to P'MACHINE_RADIX - 1). The value of this attribute is of the type <code>universal_integer</code>.</p>                                                                            |
| P'MACHINE_OVERFLOW | <p>For a prefix P that denotes a real type or subtype:</p> <p>Yields the value <code>TRUE</code> if every predefined operation on values of the base type of P either provides a correct result, or raises the exception <code>NUMERIC_ERROR</code> in overflow situations; yields the value <code>FALSE</code> otherwise. The value of this attribute is of the predefined type <code>BOOLEAN</code>.</p> |
| P'MACHINE_RADIX    | <p>For a prefix P that denotes a floating-point type or subtype:</p> <p>Yields the value of the radix used by the machine representation of the base type of P. The value of this attribute is of the type <code>universal_integer</code>.</p>                                                                                                                                                             |
| P'MACHINE_ROUND    | <p>For a prefix P that denote a real type or subtype:</p> <p>Yields the value <code>TRUE</code> if every predefined arithmetic operation on values of the base type of P either returns an exact result or performs rounding; yields the value <code>FALSE</code> otherwise. The value of this attribute is of the predefined type <code>BOOLEAN</code>.</p>                                               |
| P'MANTISSA         | <p>For a prefix P that denotes a real subtype:</p> <p>Yields the number of binary digits in the binary mantissa of model numbers of the subtype P. The value of this attribute is of the type <code>universal_integer</code>.</p>                                                                                                                                                                          |



| <i>Attribute</i> | <i>Definition</i>                                                                                                                                                                                                                                                                                                                                                                                       |
|------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| P'POS            | <p>For a prefix P that denotes a discrete type or subtype:</p> <p>This attribute is a function with a single parameter. The actual parameter X must be a value of the base type of P. The result type is the type <code>universal_integer</code>. The result is the position number of the value of the actual parameter.</p>                                                                           |
| P'POSITION       | <p>For a prefix P that denotes a component of a record object:</p> <p>Yields the offset, from the start of the first storage unit occupied by the record, of the first of the storage units occupied by the component. This offset is measured in storage units. The value of this attribute is of the type <code>universal_integer</code>.</p>                                                         |
| P'PRED           | <p>For a prefix P that denotes a discrete type or subtype:</p> <p>This attribute is a function with a single parameter. The actual parameter X must be a value of the base type of P. The result type is the base type of P. The result is the value whose position number is one less than that of X. The exception <code>CONSTRAINT_ERROR</code> is raised if X equals <code>P'BASE'FIRST</code>.</p> |
| P'RANGE          | <p>For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype:</p> <p>Yields the first index range of P, that is, the range <code>P'FIRST .. P'LAST</code>.</p>                                                                                                                                                                                                  |
| P'RANGE(N)       | <p>For a prefix P that is appropriate for an array type, or that denotes a constrained array subtype: Yields the N-th index range of P, that is, the range <code>P'FIRST(N) .. P'LAST(N)</code>.</p>                                                                                                                                                                                                    |
| P'SAFE_EMAX      | <p>For a prefix P that denotes a floating-point type or subtype:</p> <p>Yields the largest exponent value in the binary canonical form of safe numbers of the base type of P. The value of this attribute is of the type <code>universal_integer</code>.</p>                                                                                                                                            |
| P'SAFE_LARGE     | <p>For a prefix P that denotes a real type or subtype:</p> <p>Yields the largest positive safe number of the base type of P. The value of this attribute is of the type <code>universal_real</code>.</p>                                                                                                                                                                                                |
| P'SAFE_SMALL     | <p>For a prefix P that denotes a real type or subtype:</p> <p>Yields the smallest positive (nonzero) safe number of the base type of P. The value of this attribute is of the type <code>universal_real</code>.</p>                                                                                                                                                                                     |

| <i>Attribute</i> | <i>Definition</i>                                                                                                                                                                                                                                                                                                                                                                                         |
|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| P'SIZE           | <p>For a prefix P that denotes an object:</p> <p>Yields the number of bits allocated to hold the object. The value of this attribute is of the type <code>universal_integer</code>.</p>                                                                                                                                                                                                                   |
| P'SIZE           | <p>For a prefix P that denotes any type or subtype:</p> <p>Yields the minimum number of bits that is needed by the implementation to hold any possible object of the type or subtype P. The value of this attribute is of the type <code>universal_integer</code>.</p>                                                                                                                                    |
| P'SMALL          | <p>For a prefix P that denotes a real subtype:</p> <p>Yields the smallest positive (nonzero) model number of the subtype P. The value of this attribute is of the type <code>universal_real</code>.</p>                                                                                                                                                                                                   |
| P'SORAGE_SIZE    | <p>For a prefix P that denotes an access type or subtype:</p> <p>Yields the total number of storage units reserved for the collection associated with the base type of P. The value of this attribute is of the type <code>universal_integer</code>.</p>                                                                                                                                                  |
| P'SORAGE_SIZE    | <p>For a prefix P that denotes a task type or a task object:</p> <p>Yields the number of storage units reserved for each activation of a task of the type P or for the activation of the task object P. The value of this attribute is of the type <code>universal_integer</code>.</p>                                                                                                                    |
| P'SUCC           | <p>For a prefix P that denotes a discrete type or subtype:</p> <p>This attribute is a function with a single parameter. The actual parameter X must be a value of the base type of P. The result type is the base type of P. The result is the value whose position number is one greater than that of X. The exception <code>CONSTRAINT_ERROR</code> is raised if X equals <code>P'BASE'LAST</code>.</p> |
| P'TERMINATED     | <p>For a prefix P that is appropriate for a task type:</p> <p>Yields the value <code>TRUE</code> if the task P is terminated; yields the value <code>FALSE</code> otherwise. The value of this attribute is of the predefined type <code>BOOLEAN</code>.</p>                                                                                                                                              |
| P'VAL            | <p>For a prefix P that denotes a discrete type or subtype:</p> <p>This attribute is a special function with a single parameter X which can be of any integer type. The result type is the base type of P. The result is the value whose position number is the <code>universal_integer</code> value corresponding to X.</p>                                                                               |

| <i>Attribute</i> | <i>Definition</i>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                  | <p>The exception <code>CONSTRAINT_ERROR</code> is raised if the <code>universal_integer</code> value corresponding to <code>X</code> is not in the range <code>P'POS(P'BASE'FIRST) .. P'POS(P'BASE'LAST)</code>.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <b>P'VALUE</b>   | <p>For a prefix <code>P</code> that denotes discrete type or subtype:</p> <p>This attribute is a function with a single parameter. The actual parameter <code>X</code> must be a value of the predefined type <code>STRING</code>. The result type is the base type of <code>P</code>. Any leading and any trailing spaces of the sequence of characters that corresponds to <code>X</code> are ignored. For an enumeration type, if the sequence of characters has the syntax of an enumeration literal and if this literal exists for the base type of <code>P</code>, the result is the corresponding enumeration value. For an integer type, if the sequence of characters has the syntax of an integer literal, with an optional single leading character that is a plus or minus sign, and if there is a corresponding value in the base type of <code>P</code>, the result is this value. In any other case, the exception <code>CONSTRAINT_ERROR</code> is raised.</p> |
| <b>P'WIDTH</b>   | <p>For a prefix <code>P</code> that denotes a discrete subtype:</p> <p>Yields the maximum image length over all values of the subtype <code>P</code> (the image is the sequence of characters returned by the attribute <code>IMAGE</code>). The value of this attribute is of the type <code>universal_integer</code>.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |

# Appendix J— Separate Compilation of Ada Units

A programming language would not be suitable for team programming if the team had to share one source file. Ada provides facilities for compiling different parts of a program separately. In this appendix we look at Ada's facilities for separate compilation.

## Compilation Units

|                                                                                      |
|--------------------------------------------------------------------------------------|
| <b>Compilation</b>                                                                   |
| A source file containing one or more compilation units submitted to an Ada compiler. |
| <b>Compilation Unit</b>                                                              |
| A portion of an Ada program that can be compiled by itself.                          |

An Ada program is a collection of compilation units submitted to a compiler in one or more compilations. A **compilation** is the Ada term for a source file submitted to an Ada compiler. A portion of an Ada program that can be compiled by itself is called a **compilation unit**. A compilation contains one or more compilation units.

For greatest flexibility, we restrict our examples to one compilation unit per compilation and recommend that you do the same. Ada does not specify any rules for naming our source files. Each different Ada development environment has its own conventions. Check the manual that comes with your Ada compiler to learn how to name the files in which you store your Ada code.

Here is a simplified EBNF definition that defines what portions of an Ada program may be compiled by themselves.

|                   |     |                                                                                                 |
|-------------------|-----|-------------------------------------------------------------------------------------------------|
| compilation_unit  | ::= | context_clause library_unit  <br>context_clause secondary_unit                                  |
| context_clause    | ::= | {with_clause {use_clause} }                                                                     |
| library_unit      | ::= | package_declaration  <br>subprogram_declaration  <br>subprogram_body  <br>generic_instantiation |
| secondary_unit    | ::= | library_unit_body   subunit                                                                     |
| library_unit_body | ::= | subprogram_body   package_body                                                                  |

A compilation unit may contain zero or more with clauses. These clauses tell what other units this compilation unit depends on. For example, many of our compilation units depend on Text\_IO. It has probably become an automatic reflex for you to type `with Text_IO;` each time you begin a new program. The with clause makes a library unit available within the compilation unit. This means, for example, that once we have included Text\_IO in a context clause, Text\_IO may be used anywhere within the compilation unit.

It is important to distinguish between *library units* and *secondary units*. Library units are specifications. Only library units may be referenced in a with clause. Package declara-

tions, subprogram declarations, generic instantiations, and some subprogram bodies are library units.

A secondary unit defines the body of a library unit, or in the case of a subunit defines the proper body of a subprogram unit that is declared within another compilation unit. Package bodies, subunits, and some subprogram bodies are secondary units.

What determines whether a subprogram body is a library unit or a secondary unit? The declaration of a subprogram is optional. The subprogram specification of the subprogram body acts as the declaration in the absence of a separate declaration. Therefore, a subprogram body is a library unit if there is no previously compiled subprogram declaration. A subprogram body is a secondary unit if a separate subprogram declaration has already been compiled. The following table summarizes the classification of compilation units.

| <i>Library Units</i>                                           | <i>Secondary Units</i>                                           |
|----------------------------------------------------------------|------------------------------------------------------------------|
| Package Declarations                                           | Package Bodies                                                   |
| Subprogram Declarations                                        | Subprogram Bodies (if its declaration has already been compiled) |
| Subprogram Bodies (if no declaration for it has been compiled) | Subunits (indicated by the separate clause)                      |
| Package Instantiations                                         |                                                                  |

### The Program Library

Ada requires that the language rules be enforced in the same manner for a program made up of multiple compilations as for a single compilation program. This requirement means, for example, that whether a procedure is defined within the main program file or in an independently compiled package declaration, the Ada compiler must ensure that the actual parameters of any calls to the procedure are compatible with the formal parameters. To enforce the language rules for programs composed of multiple compilation units, the Ada compiler uses a program library to keep track of all of the units making up a program.

Whenever a compilation unit is compiled successfully, information about it is added to the program library. Ada does not specify the exact nature of the information stored in the library for each compilation unit. The information varies from system to system and may include

1. The unit name and location of the source file containing the unit.
2. The date and time the unit was compiled.
3. A list of all units that this unit depends on (named by with clauses).
4. A *symbol table* for library units. (A list of identifiers declared in the library unit that may be used by other units. It includes a formal description of each identifier.)
5. The object code created for secondary units.

Each program uses a single library. This library contains the information for each of the compilation units making up that program. One library may be used to store the information for the compilation units making up several different programs. The relationship between programs and program libraries is illustrated in Figure J-1. In this figure, Library A contains information on the compilation units making up programs 1, 2, and 3. Library B contains information on the compilation units making up programs 4 and 5.



Figure J-1  
Ada Program Libraries

The programming environment in which an Ada compiler is used includes commands for manipulating program libraries. Although each Ada development environment is different, all include commands for creating a new program library and for interrogating the status of the units in a library. Many environments also include commands for copying the information on a compilation unit in one library into another library.

When a new program library is created, the predefined library units `Text_IO`, `Direct_IO`, `Sequential_IO`, `Calendar`, and others that we have not discussed are placed into the program library. Figure J-2 illustrates a newly created program library.

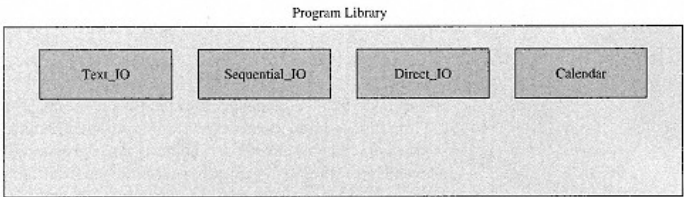


Figure J-2  
A Newly Created Program Library

When a compilation unit is submitted to the Ada compiler, the program library is accessed as part of the process of checking the validity of the new code. For example, if `Text_IO.Put` is used in the compilation unit, the Ada compiler checks the `Text_IO` unit entry in the program library to ensure that the `Text_IO.Put` parameter list is valid. If the unit is compiled successfully, it is added to the library. Figure J-3 illustrates the same library after compiling the package declaration `Pixel`, and Figure J-4 illustrates the possible contents of library unit `Pixel` just added to the program library.

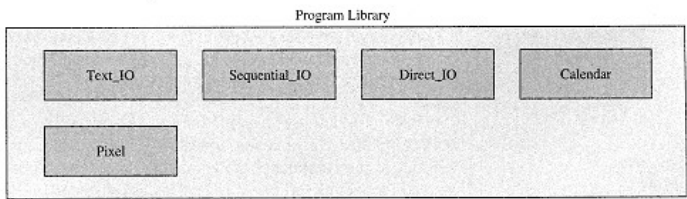


Figure J-3  
Program Library After Compiling the Package Declaration `Pixel`

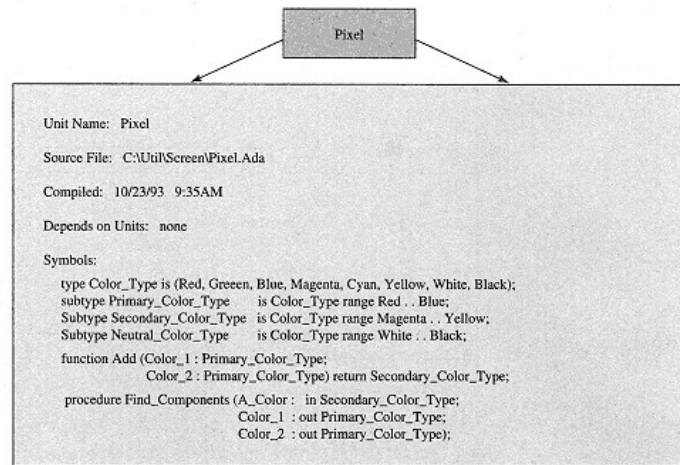


Figure J-4  
Possible Contents of Library Unit Pixel

### Order of Compilation

The following rules define the order in which units must be compiled. They are based on the visibility of units established by context (with) clauses.

1. A compilation unit cannot be compiled until all library units named in its context clause have been compiled.
2. A secondary unit that is a package body or subprogram body cannot be compiled until its corresponding library unit (its declaration) has been compiled.
3. A secondary unit that is a subunit of a parent compilation unit cannot be compiled until its parent unit has been compiled.

Let's look at an example. Each of the following boxes represents a single file to be submitted to the compiler (a compilation). Each of these compilations contains a single compilation unit. Remember that package declarations are library units and package bodies are secondary units.

```

with Apple;
with Pear;
with Text_IO;
procedure Fruit is
 .
 .
 .
end Fruit;
-- This is a library unit since there
-- is no declaration of procedure Fruit.
-- This is the main program.

```

```

package Apple is -- This is a package declaration.
 .
 .
 .
end Apple;

with Banana;
package body Apple is -- This is a package body.
 .
 .
 .
end Apple;

package Banana is -- This is a package declaration.
 .
 .
 .
end Banana;

package body Banana is -- This is a package body.
 .
 .
 .
end Banana;

with Apple;
package Pear is -- This is a package declaration.
 .
 .
 .
end Pear;

package body Pear is -- This is a package body.
 .
 .
 .
end Pear;

```

Compilation unit `Fruit` depends upon three library units: `Apple`, `Pear`, and `Text_IO`. All three of these library units must be compiled before `Fruit` can be compiled. Because `Text_IO` is a predefined package, it is already compiled and in the program library.



The library units (package declarations) `Apple`, `Banana`, and `Pear` must be compiled before their corresponding secondary units (package bodies). Because there are no `with` clauses in library units `Apple` and `Banana`, they do not depend on any other library units. However, library unit `Pear` is dependent on library unit `Apple`. Therefore library unit `Apple` must be compiled before library unit `Pear` can be compiled.

Secondary unit `Apple` depends on library unit `Banana`. `Apple`'s body cannot be compiled until `Banana`'s declaration has been compiled. No other secondary units have `with` clauses.

Figure J-5 is a dependency graph that is used to illustrate the dependencies among all the compilation units in this example. Each arrow indicates a dependency on the unit to which it points. A package is represented by a circle with the declaration and body shown as upper and lower halves. A procedure is represented by a rectangle.

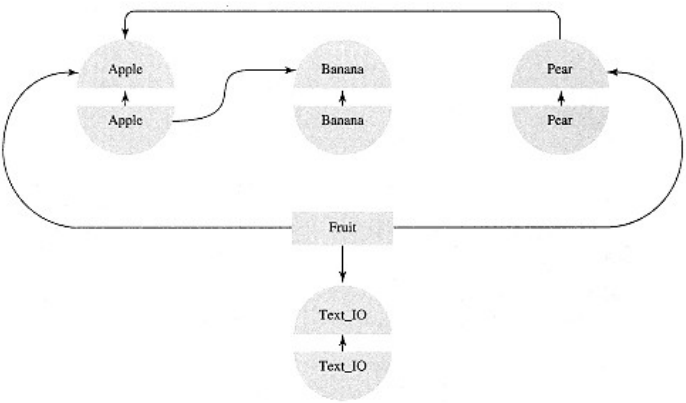


Figure J-5  
The Dependencies Among the Units Making up an Ada Program

Given the restrictions placed on the order of compilation by the dependencies specified in the `with` clauses and shown in the dependency graph of Figure J-5, the compilation order given in the following three lists of units are all valid.

| <i>Order A</i>          | <i>Order B</i>          | <i>Order C</i>          |
|-------------------------|-------------------------|-------------------------|
| 1. Banana (declaration) | 1. Apple (declaration)  | 1. Banana (declaration) |
| 2. Banana (body)        | 2. Pear (declaration)   | 2. Apple (declaration)  |
| 3. Apple (declaration)  | 3. Banana (declaration) | 3. Pear (declaration)   |
| 4. Apple (body)         | 4. Apple (body)         | 4. Fruit                |
| 5. Pear (declaration)   | 5. Pear (body)          | 5. Apple (body)         |
| 6. Pear (body)          | 6. Banana (body)        | 6. Banana (body)        |
| 7. Fruit                | 7. Fruit                | 7. Pear (body)          |

There are other valid orders of compilation for this example besides the three shown here. After all of the units have been compiled, the linking process combines the object code into the executable program. Figure J-6 illustrates this program development process.

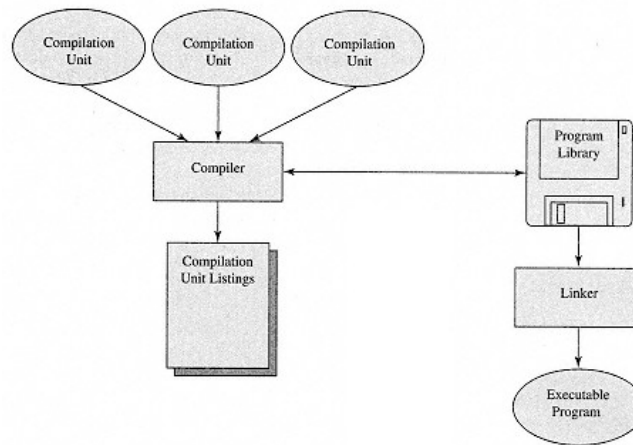


Figure J-6  
Compiling and Linking an Ada Program Made up of Multiple Units

---

#### Obsolete Unit

A compilation unit that has *not* been recompiled after a unit on which it depends has been recompiled. An error is issued if an attempt is made to use an obsolete unit to form an executable program.

---

If a library unit is changed and compiled again, all of the units that depend on it become **obsolete** and must be recompiled. The linker issues an error message if a required unit is either missing from the program library or obsolete. For example, if the package declaration for `Apple` is changed and recompiled, `Apple`'s body, `Pear`'s declaration and body, and procedure `Fruit` become obsolete; they must be recompiled. In Figure J-5 you can see that these units depend on `Apple`'s declaration. `Apple`'s body, `Pear`'s declaration, and procedure `Fruit` depend directly on `Apple`'s declaration. `Pear`'s body has an indirect dependency on `Apple`; it depends on `Pear`'s declaration that depends on `Apple`'s declaration. Dependency graphs like the one in Figure J-5 are a good way to determine what units become obsolete when another unit is recompiled.

As mentioned earlier in this appendix, the programming environment in which an Ada compiler is used includes commands for manipulating the program library. Many systems include commands for determining if any units in the library are obsolete and commands that automatically compile or recompile the units making up a program in the correct order. Such program development tools are necessary for programs with large numbers of units.

Ada 95 introduces a hierarchical library structure containing child units. Child units help reduce the number of units that become obsolete when we recompile some unit. They also allow multiple packages to share a private type. Because we have used Ada 83 in all of our examples, we do not discuss the hierarchical library and child units here. The *Ada 95 Rationale and Language Reference Manual* describe how these features provide a powerful means for the development of large systems from component subsystems.

#### Pre-Instantiation of Generic Packages

The last choice listed in our simplified EBNF definition of a library unit (on page 784) is generic instantiation. If a program needs to do input or output with `Integer` or `Float` we must use a package instantiated from the appropriate generic template (`Integer_IO`

or `Float_IO`) in package `Text_IO`. We can instantiate the numeric input/output packages we need in the declarative part of our program. Instantiations require that the compiler construct a new package from the template given in the generic package. This construction process is time consuming. Compilation time may be shortened by defining a library unit consisting of just the required generic instantiation.

Here is a file containing only two lines. These lines are all that are necessary to instantiate a package for the input and output of Integers from the generic template. These two lines are a complete compilation unit: a generic instantiation.

```
with Text_IO;
package Integer_IO is new Text_IO.Integer_IO (Num => Integer);
```

We compile this two line library unit to construct our new package from the template and place information about it into the program library. Then any other unit may include it in a `with` clause like this:

```
with Integer_IO; -- Precompiled generic instantiation
with Text_IO;
procedure Short is
begin
 Text_IO.Put ("The result is ");
 Integer_IO.Put (Item => 1815, Width => 1);
end Short;
```

For many Ada systems, space saving is a second advantage of pre-instantiation. Each new package constructed takes up space in the object file. If all programs share a single package rather than constructing one of their own, space may be saved. We recommend that you add generic instantiation library units for `Integer` and `Float` input-output to your program library and eliminate the instantiation of `Integer_IO` and `Float_IO` within each program. All of our examples assume that `Integer_IO` and `Float_IO` have been pre-instantiated into the Ada library.

## Appendix K— Random Discrete Value Generator

Random discrete values (such as random integers and random enumeration types) can be generated by instantiations of the package `Discrete_Random` listed in this appendix and included on the disk that accompanies this text. `Discrete_Random` has been tested using Meridian Ada on 386, 486, and Pentium-based microcomputers and with DEC Ada on a VAX 6340. The pseudorandom values it generates are adequate for the examples and exercises in this text. If you are using an Ada 95 compiler, you should use the package `Ada.Numerics.Discrete_Random` rather than the one given here.

The body of `Discrete_Random` uses a floating-point type with the maximum number of digits permitted by a particular Ada implementation. In some cases, the speed of calculations with this amount of precision may be too slow. To increase the speed of calculations, lower the number of digits used for type `Float_Type`. Lowering the precision reduces the allowable range of the discrete subtype (see the precondition in the package specification) and may result in a poorer distribution of pseudorandom values.

We recommend that only one generator be instantiated for a program. Multiple generators are likely to produce identical sequences of values.

Thanks to Gwilym Archer for taking time from his Simulation and Modeling class to write and test the package `Discrete_Random`.

### Package Specification

```
generic

 type Result_Subtype is (<>);

package Discrete_Random is

 -- Generates a pseudorandom value in Result_Subtype range.

 -- Preconditions : The number of values in Result_Subtype must be
 -- less than 10 ** System.Max_Digits.
 --
 -- Result_Type'Pos(Result_Type'Last) < Integer'Last

 -- If Set_Seed is not called before the first call to Random,
 -- the random generator seed is initialized from the system clock.

 -- Written by Gwilym V. Archer
 -- Department of Computer Science
 -- S.U.N.Y. Plattsburgh, Plattsburgh, N.Y.
 -- February 14, 1995

 subtype Seed_Range is Positive range 1..(Integer'Last - 1)/2;

 procedure Set_Seed(The_Seed : in Seed_Range);
 -- Sets the seed to a user-selected value in Seed_Range
```

```
-- Commonly used for debugging programs that use the discrete random
-- generator. Set the seed to the same value in different runs to
-- obtain the same sequence of random values in each run.
```

```

function Random return Result_Subtype;
-- Returns a random value in the range of Result_Subtype
```

```
end Discrete_Random;
```

### Package Body

```
with System;
with Calendar;
package body Discrete_Random is
```

```
 type Float_Type is digits System.Max_Digits;
```

```
 Seed : Float_Type; -- Seed is global to this package body
```

```

procedure Set_Seed (The_Seed : in Seed_Range) is
```

```
-- Assigns to global Seed the value of The_Seed, conditioned
-- To a positive, odd float
```

```
begin
 Seed := Float_Type((The_Seed * 2) + 1);
end Set_Seed;
```

```

function Fraction (Value : in Float_Type) return Float_Type is
```

```
-- Returns fractional part of Value.
```

```
-- Precondition: 0 <= Value < Integer'Last
```

```
begin
 if Float_Type(Integer(Value)) > Value then
 return Value - Float_Type(Integer(Value) - 1);
 else
 return Value - Float_Type(Integer(Value));
 end if;
end Fraction;
```

```

function Random return Result_Subtype is
```

```
-- Returns a pseudorandom value in the range of Result_Subtype
```

```
 Multiplier : constant Float_Type := 251.0;
```

```

Bits : constant Float_Type := Float_Type(Float_Type' Machine_Radix) **
 (Float_Type'Machine_Mantissa - 1);

-- The number of discrete values in type Result_Subtype;
Num_Values : Positive := Result_Subtype'Pos (Result_Subtype'Last) -
 Result_Subtype'Pos (Result_Subtype'First) + 1;

begin -- Function Random
 -- Increase number of digits in random number
 while Seed < Bits loop
 Seed := Seed * Multiplier;
 end loop;

 -- Shift radix point to left
 Seed := Seed/Bits;

 -- Separate fraction
 Seed := Fraction(Seed); -- Seed is a pseudorandom number greater than
 -- or equal to Zero and less than One

 -- Map random real number value to Result_Subtype range
 return Result_Subtype'Val(Integer (Seed * Float_Type(Num_Values) - 0.5)
 + Result_Subtype'Pos (Result_Subtype'First));

end Random;

begin -- Discrete_Random

 -- Get seed from system clock
 Seed := Float_Type(Calendar.Seconds(Calendar.Clock));

 -- Reduce Seed to a safe value for Integer conversion
 while Seed >= Float_Type(Integer'Last/2 - 1) loop
 Seed := Seed / Float_Type(Float_Type'Machine_Radix);
 end loop;

 -- Condition Seed to an odd-valued positive float
 Seed := Float_Type(Integer(Seed) * 2) + 1.0;

end Discrete_Random;

```

## Appendix L— Ada for Pascal Programmers

The basic features of the Ada programming language are very similar to Pascal. This appendix is meant to give a programmer familiar with Pascal a brief overview of the Ada language. We cannot give all the details of Ada nor a rationale for its features in these few pages. The *Ada Language Reference Manual* (LRM) provides a succinct description of the language. The LRM describes the concepts used in this text at a level that can be understood by an experienced Pascal programmer. Several of the other appendices in this book are taken directly from the LRM. Appendix B describes Ada syntax through a complete set of EBNF definitions. Appendixes D and E contain the specifications for the predefined input and output packages. We also recommend that you consult an introductory Ada programming textbook such as *Programming and Problem Solving in Ada* (Lexington, Mass.: D.C. Heath and Company, 1994) for additional programming examples and more detailed descriptions of basic Ada constructs.

### Example Program

We use the following program to illustrate many of the basic concepts you need to know before beginning this text. The line numbers in the left margin are keyed to explanations given in this appendix. Program `Example` is a collection of Ada constructs that serves no purpose other than to illustrate the language. We recommend that you read the explanations in the sections that follow before studying it in any detail.

```

1 with Text_IO; -- Predefined I/O package for Character and String I/O
2 with Integer_IO; -- Pre-Instantiated I/O package for Integer I/O
3 with Float_IO; -- Pre-Instantiated I/O package for Float I/O
4
5 procedure Example is
6
7 -- This program illustrates some basic Ada concepts. Most declarations
8 -- serve only as illustrations and are not used in the body of the program.
9
10
11 -- Type declarations (Scalar types)
12
13 type Count_Type is range 1..10; -- Type declaration
14 subtype Low_Count_Type is Count_Type range 1..4; -- Subtype declaration
15
16 type High_Precision is digits 15 range 0.0..5.0; -- Type declaration
17
18 subtype Response_Type is Integer range -10..10; -- Subtype declaration
19 subtype Volts is Float range 0.0..5.0; -- Subtype declaration
20
21 type Fruit_Type is (Apple, Pear, Orange, Lemon, Grapefruit); -- Enumeration type
22 subtype Citrus_Type is Fruit_Type range Orange..Grapefruit; -- Subtype declaration
23
24
25 -- Package Instantiations
26
```

```

27 -- Instantiate a package for the input and output of fruits
28 package Fruit_IO is new Text_IO Enumeration_IO (Enum => Fruit_Type);
29 -- Instantiate a package for the input and output of counts
30 package Count_IO is new Text_IO.Integer_IO (Num => Count_Type);
31 -- Instantiate a package for the input and output of high precision values
32 package Precise_IO is new Text_IO.Float_IO (Num => High_Precision);
33
34 -- Constrained array type declarations
35
36 -- One-dimensional array
37 subtype Index_Type is Integer range 1..10;
38 type Vector is array (Index_Type) of Float; -- An array of 10 Floats
39
40 -- Two-dimensional array
41 type Row_Type is range 1..10;
42 type Col_Type is range 1..100;
43 type Matrix is array (Row_Type, Col_Type) of Float;
44
45 -- Unconstrained array type declarations
46
47 type Float_Array is array (Positive range <>) of Float;
48 type Table_Type is array (Fruit_Type range <>, Character range <>) of Integer;
49
50 -- Constrained array subtypes based on unconstrained array types
51
52 subtype Velocity_Index_Type is Positive range 1..20;
53 subtype Velocity_Array is Float_Array (Velocity_Index_Type);
54
55 subtype Sales_Table is Table_Type (Fruit_Type, 'A'..'M');
56
57 subtype Name_String is String (1..10);
58
59 -- Record type with default initial values
60 type Speed_List is
61 record
62 Count : Natural := 0;
63 Values : Velocity_Array := (1..20 => 0.0);
64 end record;
65
66 -- Constant Declarations
67
68 Pi : constant := 3.1415926; -- Named number declaration
69 N : constant Integer := 5; -- Named constant declaration
70
71 -- Variable declarations
72
73 Count : Count_Type;
74 Value : Integer;
75 Setting : Float;
76 Fruit : Fruit_Type;
77
78 Thrust : Vector; -- One-dimensional array
79 Table : Matrix; -- Two-dimensional array

```



```

80 First : Name_String; -- String of 10 characters
81 Last : Name_String; -- String of 10 characters
82
83 Speed : Speed_List; -- Record variable
84
85 -----
86 procedure Get_List (List : out Speed_List) is
87 Num : Natural;
88 Value : Float;
89 begin
90 Num := 0;
91 loop
92 Float_IO.Get (Value);
93 exit when Value <= 0.0;
94 Num := Num + 1;
95 List.Values (Num) := Value;
96 end loop;
97 List.Count := Num;
98 end Get_List;
99
100 -----
101 procedure Put_Array (The_Array : in Float_Array) is
102 begin
103 for Index in The_Array'First .. The_Array'Last loop
104 Float_IO.Put (Item => The_Array(Index), Exp => 0,
105 Fore => 6, Aft => 2);
106 end loop;
107 end Put_Array;
108
109 -----
110 begin -- Program Example
111 Text_IO.Put_Line ("This is a sample program.");
112
113 for Number in 1..N loop -- Number is "declared" here
114 Integer_IO.Put (Item => Number, Width => 3);
115 end loop;
116 Text_IO.New_Line;
117
118 Text_IO.Put ("Enter a whole number: ");
119 Integer_IO.Get (Value);
120 Text_IO.Put ("Your number is ");
121 if Value < 0 then
122 Text_IO.Put_Line ("negative.");
123 elsif Value = 0 then
124 Text_IO.Put_Line ("zero.");
125 else
126 Text_IO.Put_Line ("positive.");
127 end if;
128
129 Get_List (List => Speed);
130 -- Pass a slice of the array field in the record Speed
131 Put_Array (The_Array => Speed.Values(1..Speed.Count));
132

```

```

133 Fruit_IO.Get (Fruit);
134 if Fruit < Fruit_Type'Last then
135 Fruit := Fruit_Type'Succ (Fruit);
136 else
137 Fruit := Fruit_Type'First;
138 end if;
139 Fruit_IO.Put (Item => Fruit, Width => 8, Set => Text_IO.Lower_Case);
140 end Example;

```

## Comments

Pascal uses one symbol to begin a comment and another to end the comment. In Ada, the beginning of a comment is denoted by two dashes (–); the end of the line ends the comment. "Multiple line" comments require double dashes at the beginning of each line (lines 7–8).

## Identifiers

Identifiers must begin with a letter and may contain letters, digits, and underscores. There is no maximum length; all characters are significant. Like Pascal, Ada is not case sensitive (HELLO, Hello, and hello are all the same). Programmer-defined identifiers may not be reserved words (Appendix A). See Appendix B for the formal definition of an identifier, and Appendix H for an identifier style guide.

## Declarations

Like Pascal, Ada requires that all named constants, types, variables, and subprograms be declared before they are used. Ada declarations are *not* made in sections (like Pascal's Const, Type, and Var sections). Ada allows us to make declarations of constants, types, and variables in any order. The program example above contains constant declarations (lines 68–69), type declarations (lines 11–22, and 34–64), and variable declarations (lines 71–83). Subprogram bodies come after these basic declarations.

Many declarations (lines 14, 16, 18, 19, and 22) and executable statements (line 113) use ranges. A *range* is two values separated by two periods such as 5 . . 8 and Orange . . Grapefruit. If the first value is greater than the second value, the range is considered to be null. A *null range* has no values in it. Null ranges are often encountered as the range of a `for` loop.

Both Pascal and Ada allow nesting of subprograms. The scope and visibility of names is the same in Ada as it is in Pascal. Ada's package concept is used to further restrict the scope and visibility of names. We introduce packages in Chapter 1 and we discuss them in more depth in Chapter 3.

## Types and Subtypes

Ada is much stricter with its types than Pascal. Values of different types can't be combined in the same expression. Thus while you can add an integer value and a real value together in Pascal, you cannot in Ada. The following list compares Ada's predefined types with Pascal's.

| <i>Ada</i>                    | <i>Pascal</i>        |
|-------------------------------|----------------------|
| Boolean                       | Boolean              |
| Integer                       | Integer              |
| Natural (Subtype of Integer)  | range 0..MaxInt      |
| Positive (Subtype of Integer) | range 1..MaxInt      |
| Float                         | Real                 |
| Character                     | Char                 |
| String                        | Packed Array of Char |
| Text_IO.File_Type             | Text                 |

Ada subtypes are frequently used to restrict the range of a type. For example, the predefined subtype `Positive` includes only the integer values greater than zero. Unlike different types, values of different subtypes may be combined as long as they share the same base type. Thus we can add together a `Positive` value, a `Natural` value, and an `Integer` value. Examples of programmer-defined subtypes can be found on lines 14, 18, 19, 22, 37, and 52 to 57.

When a value is assigned to a variable, Ada ensures that it's in the proper range. If it isn't in the range, a run-time error (`CONSTRAINT_ERROR`) occurs. For example, assigning -1 to a `Natural` variable would result in a run-time error.

Declaring our own types and subtypes allows our program to model our problems more accurately. This makes it easier for another programmer to understand our program and for the Ada compiler to detect any errors that we make. For example, adding gallons of fuel and cups of coffee together is acceptable if both are type `Integer`, but results in a compilation error if we declare each as a different type. Program `Example` contains examples of both type and subtype declarations. The syntax for type and subtype declarations is given in Appendix B.

### Attributes

Attributes allow us to determine the value of some property of a type or variable. Some attributes require arguments that are enclosed in parentheses. Appendix I lists all the available attributes. Attributes are written using a type or variable name and an attribute name separated by an apostrophe (a single quote). Here are some examples based on the declarations given in the sample program.

| <i>Attribute</i>      | <i>Value</i>      | <i>Comments</i>             |
|-----------------------|-------------------|-----------------------------|
| Count_Type'First      | 1                 |                             |
| Count_Type'Last       | 10                |                             |
| Integer'First         | Machine dependent | Like Pascal's -MaxInt       |
| Integer'Last          | Machine dependent | Like Pascal's +MaxInt       |
| Positive'First        | 1                 |                             |
| Volts'Last            | 5.0               |                             |
| Fruit_Type'First      | Apple             |                             |
| Fruit_Type'Last       | Grapefruit        |                             |
| Fruit_Type'Pred(Pear) | Apple             | Like Pascal's Pred function |
| Fruit_Type'Succ(Pear) | Orange            | Like Pascal's Succ function |

| <i>Attribute</i>            | <i>Value</i>     | <i>Comments</i>            |
|-----------------------------|------------------|----------------------------|
| Fruit_Type'Succ(Grapefruit) | CONSTRAINT_ERROR | Run-time error             |
| Fruit_Type'Pos(Apple)       | 0                | Like Pascal's Ord function |
| Fruit_Type'Pos(Grapefruit)  | 4                |                            |
| Fruit_Type'Val(1)           | Pear             | Val is the inverse of Pos  |
| Fruit_Type'Val(4)           | Grapefruit       |                            |
| Fruit_Type'Val(5)           | CONSTRAINT_ERROR | Run-time error             |
| Vector'First                | 1                | First index in array type  |
| Vector'Last                 | 10               | Last index in array type   |
| Vector'Length               | 10               | Number of components       |
| Vector'Range                | 1..10            | Index range                |

### Record Types

Ada record types are very similar to Pascal record types. Here, for example, is the declaration of a simple inventory record.

```
subtype Part_String is String (1..10); -- 10 character part name

type Inventory_Rec is
 record
 Part_Name : Part_String;
 Quantity : Natural;
 Price : Float;
 end record;
```

Accessing individual record fields is accomplished using the same dot notation as Pascal. Record variables can be assigned just as in Pascal. But unlike Pascal, Ada allows two records to be compared for equality and inequality using the = and /= operators.

### Record Aggregates

Record aggregates are often used to assign values to records. In its simplest form, a record aggregate is a literal. If `Part` is type `Inventory_Rec`, the following two equivalent assignment statements assign the aggregate value to `Part`.

```
Part := ("A123456789", 250, 56.95); -- These two assignment
Part := (Part_Name => "A123456789", -- statements are
 Price => 56.95, -- equivalent.
 Quantity => 250);
```

In the first assignment statement, the values in the record aggregate are assigned to the fields in `Part` according to their position in the aggregate. "A123456789" is assigned to the first field (`Part_Name`), 250 is assigned to the second field (`Quantity`), and 56.95 is assigned to the third field (`Price`). In the second assignment statement, the values in the aggregate are associated with field names. Using named association allows us to list the

values in any order. The additional documentation provided by named association often makes a program easier to read; there is no need to search for the record declaration to find what fields it contains.

### ***Initial Values***

Record type declarations may also include initial values that are assigned to each variable of that type during elaboration (we discuss elaboration later). The record type `Speed_List` (line 60) in program `Example` contains an initial value of zero for the count of items currently in the list and for all of the components of the array (using an array aggregate). The `Count` and `Values` fields in all variables of this type are initialized to zero.

### **Array Types**

Ada's arrays are used just like Pascal's except that parentheses are used around array indices (subscripts) rather than square brackets. There are many ways to declare an array type in Ada. The complete syntax for declaring array types is given in Appendix B. An array type is either constrained or unconstrained.

#### ***Constrained Arrays***

Ada's constrained arrays are just like Pascal's arrays. The number of components in the array is set in the declaration. The array types `Vector` (line 38) and `Matrix` (line 43) in program `Example` are constrained array types. Using different types (not subtypes) for the array indices in the two dimensional array `Matrix` ensures that we do not inadvertently mix up row and column subscripts. Ada's strong typing helps prevent errors that are easy to make in Pascal.

#### ***Unconstrained Arrays***

An unconstrained array type is an array type in which the number of components is not specified. Unconstrained array types are most commonly used as base types for constrained array subtypes and with subprogram parameters so that the formal parameter can match different size arrays. Type `Float_Array` (line 47) in program `Example` is a one-dimensional unconstrained array of `Float` values. The index type is `Positive`, but no range is given. Instead, the box symbol (`<>`) indicates that the range of the index will be supplied later. Type `Table_Type` (line 48) is an unconstrained two-dimensional array type whose first index is type `Fruit_Type` and whose second index is type `Character`.

In program `Example`, `Velocity_Array` (line 53) is the declaration of a constrained array type with 20 components based on the unconstrained array type `Float_Array`. `Sales_Table` (line 55) is a constrained two-dimensional array with 5 rows and 13 columns.

Unconstrained array types cannot be used in the declaration of a variable; all array variables must be constrained. However, as we show later, array subprogram parameters are usually unconstrained array types.

Finally, we note that the predefined type `String` is an unconstrained array with `Positive` index and `Character` components. The declaration of type `String` (and all the predefined types) may be found in package `Standard` (Appendix G). Type

`Name_String` (line 57) in program `Example` is a constrained array of characters based on the predefined unconstrained array type `String`.

### ***Array Operations***

Ada allows us to do much more with an array than Pascal does. We can compare two arrays for equality and inequality (`=` and `/=`). Also, if the array components are a discrete type, two arrays can be compared with any of the relational operators (`<`, `<=`, `>`, and `>=`). The relational operators are most commonly used with arrays of characters (type `String`).

Array slices allow us to manipulate portions of an array. We specify a slice by using a range rather than an index in the parentheses following the array variable name. An array slice is treated like any other array. The call to procedure `Put_Array` (line 131) in program `Example` uses an array slice to pass only the portion of the array containing data to the procedure.

Array slices are also common in character string manipulation, usually along with the catenation operator (`&`). Here is an example using the string variables `Last` and `First` declared on lines 80 and 81 of program `Example`:

```
Last(1..8) & ", " & First(1..4) -- Combine portions of last
 -- and first names
```

### ***Array Attributes***

We use array attributes to obtain information about an array type, an array variable, or an array parameter. The most common attributes used with arrays are `'First`, `'Last`, `'Length`, and `'Range`. The table on page 800 includes examples of these four array attributes with the array type `Vector` defined on line 38 of our sample program.

We commonly use array attributes with unconstrained array parameters. On line 131 of our sample program, the main program passes a slice of an array to procedure `Put_Array`. Procedure `Put_Array` (lines 101–107) uses the array attributes `'First` and `'Last` to determine the beginning and ending indexes needed to display all of the values in the unconstrained array parameter `The_Array`. Alternatively, we could use the `'Range` attribute in the `for` loop of this procedure like this:

```
for Index in The_Array'Range loop
```

To design for reuse a subprogram that has unconstrained array parameters, you should always use array attributes instead of literals. For example, we could use the following `for` loop in procedure `Put_Array`:

```
for Index in 1..The_Array'Last loop
```

However, with this loop our procedure raises `CONSTRAINT_ERROR` when the first index of the actual array parameter is greater than one. This situation is common when the actual parameter is a slice of an array as illustrated by the following call:

```
-- Display the 5th through 12th values in the list
Put_Array (The_Array => Speed.Values (5..12));
```

## Array Aggregates

Array aggregates are used to assign a value to an array variable. In its simplest form, an array aggregate is a literal. `Thrust` (line 78) is an array of 10 `Float` components (line 38). The following assignment statements use different forms of array aggregates to assign values to all the components in `Thrust`.

```
-- Aggregate using positional association
Thrust := (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0);

-- Aggregates with named association
Thrust := (1..10 => 0.0);
Thrust := (Index_Type => 0.0);
Thrust := (1 | 3 | 5 | 7 | 9 => 1.0, 2 | 4 | 6 | 8 | 10 => 2.0);
Thrust := (1..3 => 0.0, 5..9 => 1.0, others => 2.0);
Thrust := (others => 0.0);
```

As with record aggregates, the values given in an array aggregate need not be literals. The following assignment statement assigns the value of the `Float` variable `Setting` (line 75) to all 10 components of `Thrust`.

```
Thrust := (1..10 => Setting);
```

## Operator Precedence

Ada expressions are very similar to Pascal expressions. However, the precedence rules are a little different. Ada's operators are divided into six levels of precedence. The operators at each level have the same precedence. These levels, from highest precedence to lowest, are

|                         |                |
|-------------------------|----------------|
| Highest Level Operators | ** abs not     |
| Multiplying Operators   | * / rem        |
| Unary Adding Operators  | + -            |
| Binary Adding Operators | + - &          |
| Relational Operators    | = /= < <= > >= |
| Logical Operators       | and or xor     |

As with Pascal, whenever an expression contains two operators of equal precedence level, they are evaluated from left to right and parentheses may be used to override precedence. Ada *requires* parentheses when different logical operators are used in the same expression.

## Numeric Type Conversion

It is sometimes necessary to convert one numeric type to another so that they may be combined in the same arithmetic expression. Any numeric type may be converted to any other numeric type by using the target type name like a function. When a real type (like `Float`) is converted to an integer type, the value is rounded to the nearest whole number. Here, for example, is an expression that calculates an average by dividing a sum of `Float` values by the number of values (an `Integer`):

Sum / Float (N)

The value of  $N$  is not changed by this expression; the type name, `Float`, acts like a function that returns the conversion results.

## Statements

Each statement is *terminated* by a semicolon (in Pascal, statements are *separated* by semicolons). The syntax of all statements is given in Appendix B. In this section, we make brief comments and give some examples.

### Assignment Statement

Ada's assignment statement is identical to Pascal's.

### If Statement

If statements are terminated by `end if`. This bracketing end eliminates the need for the compound statement (`begin..end`) required in Pascal when you need to execute more than one statement for a particular alternative. An if statement may contain zero or more `elsif` clauses and zero or one `else` clause. Lines 121–127 and 134–138 of program `Example` illustrate Ada's if statement.

### Case Statement

When a decision is made on the basis of a discrete value, the case statement is usually preferred over the if statement. The major difference between Pascal's case statement and Ada's case statement is that with Ada, every possible value that the case selector can take *must* be listed. You can use an `others` clause as the last case to handle any values not already handled. Here is an example using a character case selector.

```
case Char_Variable is
 when ' ' =>
 Text_IO.Put_Line ("Blank.");
 when 'a'..'z' =>
 Text_IO.Put_Line ("Lowercase.");
 when 'A'..'Z' =>
 Text_IO.Put_Line ("Uppercase.");
 when '.', '!' | '?' | '!' =>
 Text_IO.Put_Line ("Sentence Terminator.");
 when others =>
 Text_IO.Put_Line ("Some other character.");
end case;
```

This example has just a single statement to execute for each of the five case alternatives, but each alternative may have any number of statements.



## Loop Statements

Ada has three different loop constructs. All loops may have an optional label to facilitate matching ends in nested loops.

Ada's while loop is just like Pascal's.

The `loop` and `exit` statements are combined to make pretest, midtest, and posttest loops. These loops terminate when the exit condition is true. Here are examples of pretest, midtest, and posttest loops. We have included an optional label in each.

```
Pre: Mid: Post:
loop loop loop
 exit Pre when A > 5; . .
 . . .
 . . .
 . exit Mid when A > 5; exit Post when A > 5;
end loop Pre; . end loop Post;
 .
 .
 end loop Mid;
```

Ada's for loop is similar, but not identical to Pascal's for loop. One big difference is that the loop control variable should not be declared before the loop. The loop control variable is "declared" in the for loop statement. Line 113 of program `Example` contains a for loop. The loop control variable `Number` is not declared prior to this statement. The loop control variable is not available outside of the loop, and it cannot be changed within the loop body.

The loop control variable goes through all the values in the specified range from smallest to largest unless `reverse` is specified, in which case the loop control variable goes from the largest to the smallest value. The loop is not executed if the range is null, regardless of whether or not `reverse` is specified. Study the following examples to ensure that you understand this loop statement.

```
for I in 1..5 loop
 Integer_IO.Put (I); -- Displays 1 through 5
end loop;

for I in reverse 1..5 loop
 Integer_IO.Put (I); -- Displays 5 through 1
end loop;

for I in 5..1 loop
 Integer_IO.Put (I); -- Loop body not executed (null range)
end loop;

for I in reverse 5..1 loop
 Integer_IO.Put (I); -- Loop body not executed (null range)
end loop;
```

Subprograms

There are some significant differences and several minor differences between Pascal and Ada subprograms. We discuss these in the next sections.

Parameter Association

The association between actual parameters and formal parameters can be made *by name* or *by position*. Pascal only allows association by position. Here are three "identical" calls to a procedure that calculates the area of a rectangle:

```
Calc_Area 4, 3, Result); -- Positional Association
Calc_Area Width => 4, Height => 3, Area => Result); -- Named Association
Calc_Area Area => Result, Height => 3, Width => 4); -- Named Association
```

With named association, the name of the *formal parameter* is given on the left of the arrow symbol (=>) and the name (or expression) of the *actual parameter* on the right of this symbol. Named association allows us to list the parameters in any order. Named association usually makes a program easier to read; you don't have to keep flipping between the procedure call and the procedure itself to see how the parameters match.

We tend to use positional association for calling procedures with a single parameter and with function subprograms. Otherwise we prefer to use named association because the extra documentation provided outweighs the effort needed to produce it.

Parameter Modes

Pascal has two parameter modes: VAR (by reference) and non-VAR (by value). These modes are distinguished by the parameter passing *mechanism*. Ada has three parameter modes: *in*, *out*, and *in out*. In Ada, we choose the mode based on how the parameter is used by the procedure:

| Mode   | Use                                                                                 | Restrictions Enforced by Ada                                                                                                     |
|--------|-------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------|
| in     | Passing information into the subprogram                                             | Subprogram cannot change the parameter.                                                                                          |
| out    | Passing results out of the procedure                                                | The procedure may only assign a value to the parameter. It may not use it in any other way. (This restriction lifted in Ada 95.) |
| in out | Passing a value to the procedure that is changed by the procedure and then returned | None                                                                                                                             |

The actual parameter for *in* mode may be any expression. Because *out* and *in out* mode parameters return a value, the actual parameter must be a variable.

The Ada compiler selects the most efficient mechanism for passing values based on the type of the value. Programmers should be concerned about the direction of information flow and not with how the values are passed. The restrictions imposed by Ada on the use of *in* and *out* parameters within the subprogram may be overcome through the use of local variables. Here is an Ada procedure to calculate the area of a rectangle that illustrates parameter modes:

```

procedure Calc_Area (Width : in Positive;
 Height : in Positive;
 Area : out Positive) is
begin
 Area := Width * Height;
end Area;

```

Program Example contains additional examples of procedures (lines 86 and 101).

### ***Overloading***

We may use the same subprogram name over and over as long as the parameter profile (number of parameters and types of parameters) differ. The following procedure returns the area of a rectangle. It differs from the example in the previous section in that the parameters are type `Float`. But because it serves the same purpose we use the same name.

```

procedure Calc_Area (Width : in Float;
 Height : in Float;
 Area : out Float) is
begin
 Area := Width * Height;
end Area;

```

If we call procedure `Calc_Area` with `Positive` actual parameters, the first procedure is called. If we call procedure `Calc_Area` with `Float` actual parameters, the second procedure is called. If we call procedure `Calc_Area` with a mixture of `Positive` and `Float` actual parameters, the Ada compiler will tell us that this is a syntax error (unless we have defined a `Calc_Area` procedure with this parameter profile).

### ***Function Subprograms***

Ada functions may return any type (including array and record types). Pascal functions are restricted to returning scalar types. There are also three minor differences between Pascal and Ada function subprograms. First, all function parameters must be mode `in`. Second, function results are returned via a return statement rather than assigning a value to the function's name. The return statement is illustrated in the following function to calculate the area of a rectangle:

```

function Area (Width : in Positive;
 Height : in Positive) return Positive is
 Result : Positive;
begin
 Result := Width * Height;
 return Result;
end Area;

```

Lastly, we can use operator names, enclosed in quotation marks, as function names. This gives us the ability to overload Ada's operators for our own data types. In the following example we define a complex number type and an appropriate addition operator for adding two complex numbers.

```

type Complex is
 record
 Real : Float;
 Imaginary : Float;
 end record;

A : Complex;
B : Complex;
C : Complex;

function "+" (Left : in Complex; Right : in Complex) return Complex is
 Result : Complex;
begin
 Result.Real := Left.Real + Right.Real,
 Result.Imaginary := Left.Imaginary + Right.Imaginary;
 return Result;
end "+";

```

The plus operator used *within* this function is the one predefined for `Float` values. Once we have defined a new operator, we can use it like any other operator or like a function call. The following two assignment statements accomplish the same result.

```

A := B + C; -- Use the function we wrote as an operator
A := "+" (B, C); -- Normal function call

```

### Array Parameters

One of Pascal's major shortcomings is its inability to pass different size arrays to the same procedure. Ada's ability to accomplish this feat is based on unconstrained array types. A formal unconstrained array parameter matches any actual constrained array based on the unconstrained type. The following function illustrates how this is accomplished. Type `Float_Array` is declared on line 47 of program `Example`.

```

function Average (Values : Float_Array) return Float is
 Sum : Float;
begin
 Sum := 0.0;
 for Index in Values'First .. Values'Last loop
 Sum := Sum + Values (Index);
 end loop;
 return Sum / Float (Values'Length);
end Average;

```

This function uses three array attributes. `'First` returns the index of the first component in the array (it need not be 1). `'Last` returns the index of the last component in the array. `'Length` returns the number of components in the array. Notice the use of the explicit type conversion to convert the integer count of array components to a `Float` value. The use of unconstrained array parameters and array attributes allows us to write subprograms that can process arrays of any size.

## Input/Output

Input and output are often the most confusing part of any programming language. We recommend that you spend some time writing some simple programs to experiment with Ada's I/O facilities. Unlike Pascal, Ada does not have any input or output statements. Input and output are accomplished by calling subprograms. Although it is possible to write our own input and output subprograms, we usually use predefined subprograms from the Ada library (the library is discussed in Appendix J). Input and output are complicated by Ada's strong typing. We need a different set of subprograms for each different type.

### Text Files

Package `Text_IO` contains types, subtypes, procedures, and functions for processing text files—files whose components are characters. The standard input and output files (usually the keyboard and display screen) are text files. Appendix D gives the specification of `Text_IO`. This package contains `Get` and `Put` procedures for both characters and strings; functions equivalent to Pascal's EOF and EOLN; procedures to open, create and close files; and many other useful subprograms. Even if you have already programmed in Ada, it is worthwhile to review Appendix D.

To gain access to anything in `Text_IO` we must include its name in a `with` statement as illustrated by the sample program at the beginning of this appendix. Any identifier we use from this package must be *qualified* with the package name. For example, to display the message *Hello Ada*, we use the `Put` procedure from package `Text_IO` as follows:

```
Text_IO.Put (Item => "Hello Ada");
```

Diagram illustrating the components of the statement `Text_IO.Put (Item => "Hello Ada");`:

- `Text_IO`: Package name
- `.Put`: Procedure name
- `(Item =>`: Formal parameter
- `"Hello Ada")`: Actual parameter

Qualification of names such as `Put` may be eliminated through a `use` clause. However, we believe that the documentation provided by qualification far outweighs the effort it takes to do the qualification.

### Input and Output of Enumeration Types

Pascal's inability to read and write enumeration values severely limits the usefulness of these types. Package `Text_IO` provides facilities for doing input and output with enumeration types—a generic package called `Enumeration_IO`. A generic package is a template used to construct packages that have only slight differences. In the case of the input and output of enumeration types, the only difference from one I/O package to the next is the enumeration type. This difference is supplied as a parameter in a declaration called *instantiation*. The following statement instantiates a package called `Fruit_IO` from the enumeration template in package `Text_IO`.

```
package Fruit_IO is new Text_IO.Generic_IO (Enum => Fruit_Type);
```

Diagram illustrating the components of the Ada package instantiation:

- `Fruit_IO`: Name we chose for our new package
- `Text_IO`: Package containing Enumeration\_IO
- `Generic_IO`: Generic package (template) name
- `Enum`: Generic formal parameter
- `Fruit_Type`: Generic actual parameter

We used this instantiation in line 28 of program `Example`. On lines 133–139, the program reads a fruit, then determines and displays its successor, wrapping around to the first fruit if necessary. Because `Citrus_Type` is a subtype of `Fruit_Type`, package `Fruit_IO` can be used for values of this subtype. Appendix D gives a complete specification of the operations in the generic package `Enumeration_IO`.

### Input and Output of Numeric Types

Every different numeric type requires a different set of I/O subprograms. As with enumeration types, package `Text_IO` contains generic templates for numeric types. Two of the numeric templates provided are `Integer_IO` for creating I/O packages for any integer type and `Float_IO` for creating I/O packages for any float type. Here are the two instantiations that create I/O packages for the predefined numeric types `Integer` and `Float`.

```
package Integer_IO is new Text_IO.Integer_IO (Num => Integer);
package Float_IO is new Text_IO.Float_IO (Num => Float);
```

We could have included these two declarations in program `Example`. However, because input and output of `Float` and `Integer` values is so common in our programs, we have chosen to pre-instantiate these packages. To pre-instantiate these two packages, prepare a file containing the two instantiations given above. Include the line `with Text_IO` before *each* of these instantiation lines. Compiling this file of four lines places the newly created packages (`Integer_IO` and `Float_IO`) into your Ada library. To use them in any program you include them in a `with` statement as we did in program `Example`. Nearly all the programs in this book use these two pre-instantiated I/O packages.

### Binary Files

Because binary files store values in the same form used internally by the computer, they usually are smaller and faster than storing the same information in a text file. Ada has two types of binary files: direct files and sequential files. The sequential files are equivalent to Pascal's binary files. The direct files have the added ability to access the file components in any order. The operations for these files are found in the generic packages `Direct_IO` and `Sequential_IO`. We must instantiate a different package for each different data type. The specification of these two generic packages is given in Appendix E. We discuss and use binary files in Chapters 8 and 10.

## Elaboration

Pascal programmers are taught that declarations are only for the compiler; they serve no purpose during program execution. Ada declarations are different—they are "executed." The execution of declarations is called *elaboration*. The most common use of elaboration is to initialize variables and to set the size of local array variables. The following function illustrates both of these uses.

```
function Median (Values : in Float_Array) return Float is
 Middle : Positive := (Values'First + Values'Last) / 2;
 Temp : Float_Array (Values'Range) := Values;
begin
 Sort (Temp);
 return Temp(Middle);
end Median;
```

This function returns the median value of an array of float values. To do this, it sorts the list and then returns the middle value in the sorted list. Because `Values` is an `in` mode parameter, the function cannot change it. So instead, it sorts a copy of the array. Every time `Median` is called, the local variable `Middle` is initialized to a value halfway between the first and last indices of the array `Values`. `Temp` is an array whose subscript range (and therefore size) matches that of `Values`. Every time `Median` is called, enough memory is obtained for `Temp` and it is initialized to `Values`.

## Exceptions

You probably have experience with your Pascal program crashing when you made a mistake entering your data. Another feature of Ada not available in Pascal allows us to write programs that continue to run after a run-time error occurs. Events such as division by zero or a user entering a letter when a number was requested usually crash a Pascal program, but in Ada we can write exception handlers that execute when such events occur. Exception handlers are very useful for providing robust input. Because detailed coverage of exceptions is beyond the scope of this brief introduction to Ada, we give only a simple example here. The code given on the next page uses two predefined exceptions to ensure that a positive number is entered.

```

Number : Positive;
.
.
.
Text_IO.Put_Line ("Please enter a positive whole number.");
Validation_Loop:
loop
 Validation_Block:
 begin
 Integer_IO.Get (Item => Number);
 exit Validation_Loop;
 exception
 when Text_IO.DATA_ERROR =>
 Text_IO.Put_Line ("The number must only contain digits.");
 Text_IO.Skip_Line; -- Skip over bad data
 when CONSTRAINT_ERROR =>
 Text_IO.Put_Line ("The number must be greater than zero.");
 end Validation_Block;
end Validation_Loop;

```

This code fragment contains two exception handlers: one for the exception `DATA_ERROR` defined in package `Text_IO` and one for the predefined exception `CONSTRAINT_ERROR`. These handlers are associated with the block of two statements between `begin` and `exception`. `DATA_ERROR` is raised by procedure `Get` if data is entered that is incompatible with the type. Compatible data for `Integer_IO` is any series of digits optionally preceded by a minus sign. If the user enters anything else, `DATA_ERROR` is raised. A negative number is valid data for `Integer_IO`. However, when an attempt is made to assign this negative value to the actual parameter `Number` (which is type `Positive`) `CONSTRAINT_ERROR` is raised. Only values greater than zero may be assigned to `Number`. When one of these exceptions is raised, control is transferred to the appropriate exception handler which displays a relevant error message. Execution then continues with the next loop iteration. We exit the loop when no exception is raised in the call to `Get`.

In our example, we associated exception handlers with a block statement (Ada's equivalent to Pascal's compound statement). Exceptions may also be associated with procedures and functions.

If an exception is raised and there is no local exception handler defined for it, the exception propagates back through the subprogram call chain checking for an appropriate exception handler in each subprogram. If it does not find a handler, the exception is propagated to the computer's operating system and a run-time error message is displayed and the program halted.

Besides the predefined exceptions (Appendix G) and those defined in package `Text_IO` (Appendix D), we can also declare and raise our own exceptions. We first use programmer-defined exceptions in Chapter 3. Nearly every package we write defines one or more programmer-defined exceptions.



## Glossary

### A

#### **abstract data object** (ADO)

An entity that has a state and a set of operations to examine and change its state. Also called an abstract state machine.

#### **abstract data type** (ADT)

A class of data objects with a defined set of properties and a set of operations that process the data objects while maintaining the properties.

#### **abstraction**

A model of a complex system that includes only the essential details relative to the perspective of the viewer of the system.

#### **abstract step**

An algorithmic step for which some implementation details remain unspecified.

#### **access type**

A data type that addresses or otherwise indicates the location of a variable of a given type.

#### **access type variable**

A variable that designates (provides access) to an object of a given type.

#### **ACM**

Association for Computing Machinery, a professional society of computer scientists, programmers, and others interested in computers and data processing.

#### **activation record**

A record used at run time to store information about a procedure or function call, including the parameter, local variables, register values, and return address.

#### **actual parameter**

A variable or expression contained in a procedure or function call and passed to that procedure or function.

#### **actual parameter list**

The list of actual parameters contained in a procedure or function call.

#### **address**

A label (name, number, or symbol) designating a location in memory.

#### **adjacency list**

A linked list that identifies all the vertices to which a particular vertex is connected. Each vertex has its own adjacency list.

#### **adjacency matrix**

For a graph with  $N$  nodes, an  $N \times N$  table that shows the existence (or weights) of all edges in the graph.

#### **adjacent nodes**

Two nodes in a graph that are connected by an edge.

#### **adjacent vertices**

Vertices in a graph that are connected by an edge.

#### **ADO**

*See* abstract data object.

#### **ADT**

*See* abstract data type.

#### **aggregate**

A collection of component values contained within parentheses representing the value of an array or record.

#### **alias**

One of several access variables that designate the same object.

#### **algorithm**

A step-by-step procedure for solving a problem in a finite amount of time; a verbal or written description of a logical sequence of actions.

#### **algorithm trace**

A process for verifying an algorithm by establishing a set of preconditions and postconditions for each module and then examining the algorithm step by step to check that the preconditions and postconditions are preserved.

#### **allocate**

To set aside space in memory. *See* dynamic and static allocation.

#### **alphanumeric**

A general term for human-readable alphabetic letters, numeric digits, and special characters that are interpreted with special meanings by a peripheral device. *See* graphic character and control character.

#### **ALU**

*See* arithmetic/logic unit.

#### **ancestor**

A node in a tree that is the parent of another node or the parent of another ancestor of that node.

#### **anonymous type**

A type or subtype defined in the declaration of a variable, so called because it does not have an identifier (a name) associated with it.

#### **ANSI**

American National Standards Institute, an organization that promotes voluntary standards in the United States.

#### **arc**

The connection between two vertices in a graph.

#### **argument**

*See* formal parameter and actual parameter.

#### **arithmetic logic unit** (ALU)

The computer component that performs arithmetic operations (addition, subtraction, multiplication, division) and logical operations (comparison of two values).

#### **arithmetic operator**

A symbol used in a numeric expression whose operation results in a numeric value.

#### **array**

A composite data type composed of a fixed number of components of the same type, with each component directly accessed by the index.

#### **ASCII**

American Standard Code for Information Interchange, a widely used encoding scheme for a character set composed of printable and control characters.

#### **assembler**

A program that translates an assembly language program into machine code.

**assembly language**

A language, similar to machine code, that uses mnemonics to represent operations and identifiers to represent addresses.

**assignment operator**

The Ada symbol ":= " used in an assignment statement.

**assignment statement**

A statement that gives the value of an expression to a variable.

**atomic data type**

A data type that allows only a single value to be associated with an identifier of that type; a type that is not composite; any of the Ada types Integer, Float, Boolean, Character, or any enumeration type.

**attribute**

An operator that yields a characteristic of an identifier; some attributes are functions, requiring a parameter.

**automatic range-checking**

The automatic detection of the assignment of an out-of-range value to a variable.

**auxiliary storage device**

A device that stores data in coded form, external to the computer's memory.

**B****base**

The number of digits used in a number system (for example, decimal uses 10, binary uses 2).

**base address**

The starting location in memory of a variable.

**base case**

The case for which the solution can be stated nonrecursively.

**base type**

The type from which the operations and values for a subtype are taken.

**batch processing**

A technique for executing programs and data without intermediate user interaction with the computer.

**Big-O**

*See* order of magnitude.

**binary**

Expressed in terms of combinations of the numbers 1 and 0 only.

**binary expression tree**

Binary tree in which each nonleaf node contains a binary operator and each leaf node contains one of the operands of its parent node.

**binary file**

A file data type whose components are stored in the internal binary representation of the machine; a file terminator follows the last component in the file. A binary file is created within one program to be read by another program or by the same program at a later date.

**binary operator**

An operator requiring two operands. *See* arithmetic operator, logical operator, relational operator.

**binary search**

A search algorithm for sorted lists that involves dividing the list in half and determining, by value comparison, whether the item would be in the upper or lower half; the process is performed repeatedly until either the item is found or it is determined that the item is not on the list.

**binary search tree**

A binary tree in which the key value in every node is greater than the key values in its left subtree and less than the key values in its right subtree.

**binary tree**

A data structure with a unique starting node (the root), in which each node is capable of having two child nodes and in which a unique path exists from the root to every other node.

**binding**

The association of a memory address with a variable name.

**binding time**

The point in the compile/link/execution cycle when variable names are associated with addresses in memory.

**bit**

A binary digit (1 or 0) used to represent information in a computer. Several bits make up a byte. *See* byte, word.

**black box testing**

Testing a program or subprogram based on the possible input values, treating the code as a "black box."

**body**

*See* package body or subprogram body.

**Boolean**

Data type consisting of only two values: True and False.

**Boolean expression**

An assertion that can be evaluated as being either True or False, the only values of the Boolean data type.

**Boolean operator**

*See* logical operator.

**booting the system**

The process of starting up a computer by loading the operating system into its main memory.

**bounded-length string**

A varying-length string with a fixed maximum number of characters. A bounded-length string has a current length and a maximum length.

**branch**

*See* selection.

**breadth-first search**

Searching strategy of visiting nodes one level at a time.

**bucket**

A collection of records associated with a particular hash address.

**buffer**

An intermediate data storage area usually used to balance the different operating speeds of computer components (for example, slow I/O and the faster CPU).

**bug**

An error in a program that prevents compilation or execution or causes incorrect results.

**bushy tree**

A subjective term describing the shape of a binary tree. The height of a bushy tree is closer to the minimum-height tree than the maximum-height tree.

**byte**

A sequence of bits (often 8) used to encode a character within a computer. *See* word.

**C****call**

A transfer of control from one portion of a program to a named subroutine (procedure or function).

**cancellation error**

A loss in accuracy during addition or subtraction of numbers of widely differing sizes, due to limits of precision. *See* representational error.

**cardinality**

The number of values contained in a discrete type.

**case selector**

The expression whose value determines which case alternative is selected.

**case statement**

A selection control structure that provides for multi-way selection of different courses of action.

**cast**

*See* explicit type conversion.

**cathode ray tube (CRT) screen**

An electronic tube with a screen upon which visual information may be displayed (used in computer video terminals and television sets).

**central processing unit (CPU)**

The part of the computer that executes the instructions (program) stored in memory; consists of the arithmetic/logic unit and the control unit.

**chain**

Linked list of elements that share the same hash address.

**Character**

Data type consisting of one alphanumeric character (letter, digit, or special symbol).

**character set**

The set of machine-representable characters encoded according to a specific coding system. *See* collating sequence.

**chief architect**

The leader of a programming team; determines the basic structure of the program and then delegates responsibility to implement the major modules.

**child**

The successor of a node in a binary tree. Each node may have 0, 1, or 2 children.

**circular linked list**

Linked list in which every node has a successor; the "last" element is succeeded by the "first" element.

**class**

*See* object class.

**clear box testing**

Testing a program or subprogram based on covering all of the branches or paths in the code; also called glass box testing.

**clustering**

The tendency of records to become unevenly distributed in a hash table, with many records clustering around a single address.

**code**

All or part of a program. To write all or part of a program in a programming language. *See* programming language.

**code coverage**

*See* clear box testing.

**code inspection**

Meeting of a technical team to perform a formal reading of the source code of a computer program, with the goal of identifying errors.

**code trace**

A verification process for a program in which each statement is examined to check that it faithfully implements the corresponding algorithmic step, and that the preconditions and postconditions of each module are preserved.

**code walkthrough**

Meeting of a technical team to perform a manual simulation of the source code of a computer program with sets of sample test data, with the goal of finding errors.

**coding**

Translating an algorithm into a programming language; also, the process of assigning bit patterns to pieces of information.

**collating sequence**

The ordering of a computer's character set. Ada uses the ASCII character set.

**collection class**

*See* data structure.

**collision**

Condition resulting when two or more keys produce the same hash address.

**column-major order**

An array organization where the elements are stored in memory column by column.

**comment**

A note in a program intended for human understanding but ignored by the compiler.

**comparison operator**

*See* relational operator.

**compilation**

A source file containing one or more compilation units submitted to an Ada compiler.

**compilation unit**

A portion of an Ada program that can be compiled by itself.

**compile**

To translate a program in a high-level language into machine language, using a compiler.

**compile time**

The phase of program translation (as opposed to the phase of program execution known as run time).

**compiler**

A program that translates a high-level language program (source code) into machine code (object code).

**compiler listing**

A copy of a program output by the compiler into which have been inserted messages from the compiler (indicating errors in the program that prevent its translation into machine language if appropriate).

**complete binary tree**

Binary tree that is either full or full through the next-to-last level, with the leaves on the last level as far left as possible.

**complete graph**

Graph in which every vertex is connected to every other vertex.

**component**

A logical part or element of a data structure.

**composite data type**

A data type that allows a collection of values to be associated with an identifier of that type. Arrays, records, and files are all composite types.

**computer**

A programmable electronic device that can store, retrieve, and process data.

**computer program**

A sequence of instructions outlining the steps to be performed by a computer.

**computer programming**

The process of planning a sequence of instructions for a computer to perform.

**concrete step**

A step for which the implementation details are fully specified.

**condition**

A Boolean expression used to determine the action of a selection or looping control structure.

**conditional**

*See* selection.

**constant**

An item in a program whose value is fixed at elaboration time and cannot be changed during execution. *See* literal, named constant, and named number.

**constant time**

An algorithm whose Big-O work expression is a constant: O(1).

**constrained array type**

An array type with a range specified for its index.

**constructor**

An operation that alters the state of an object.

**control abstraction**

The separation of the logical properties of a control structure from its implementation; also called procedural abstraction.

**control character**

A character that is interpreted with special meaning by a peripheral device; used to control the device.

**control structure**

A construct that determines the flow of control in part of a program and is usually represented by a statement, with the basic types being the sequence, selection, loop, and call.

**control unit**

The computer component that controls the actions of the other components in order to execute instructions (a program) in sequence.

**crash**

The cessation of a computer's operations as a result of the failure of one of its components.

**cross reference**

An index of the identifiers used in a program indicating the lines on which each was used.

**D****data**

Information that has been put into a form usable by a computer—that is, a form suitable for analysis or decision making.

**data abstraction**

The separation of the logical properties of the organization of a program's data from its implementation, ignoring inessential details.

**data structure**

A class composed of elements of other classes whose organization is characterized by accessing operations that are used to store and retrieve the individual elements. Sometimes called a collection class.

**data type**

A formal description of the set of values that a variable of a given type may take (called the *domain*) and the basic operations that can be applied to these values.

**debugging**

The task of removing errors, or "bugs," from a computer program.

**declaration**

A statement that associates an identifier with a process or object so that the user can refer to that process or object by name.

**declarative part**

A sequence of declarations in a program unit or block statement.

**declarative region**

The portion of a subprogram that begins immediately after the subprogram name and goes through the end of its body; used to determine scope of access.

**deferred constant**

A constant declared in the visible part of a package declaration whose type is a private or limited private type; the actual value of the constant is assigned in the private part of the package.

**depth-first search**

Searching strategy of going down a branch to its deepest point before examining other nodes on the same level.

**descendant**

A node in a tree that is the child of another node, or the child of another descendant of that node.

**design inspection**

Meeting of a technical team to perform a formal reading of the design of a computer program, with the goal of identifying errors.

**design walkthrough**

Meeting of a technical team to perform a manual simulation of the design of a computer program with sets of sample test data, with the goal of finding errors.

**designator**

The name that identifies a function; may be an identifier or operator symbol.

**deskchecking**

A verification technique whereby the programmer writes down essential data then walks through the program marking any changes in the data to see if the results are as expected.

**digraph**

*See* directed graph.

**direct access**

*See* random access.

**direct file**

A binary file whose components can be accessed sequentially or randomly (in any order).

**directed graph**

Graph in which each edge is directed from one vertex to one (may be the same) vertex; sometimes called a digraph.

**discrete data type**

An atomic data type in which each value (except the first) has a unique predecessor and each value (except the last) has a unique successor.

**disk**

A secondary mass storage medium providing a large amount of permanent storage; a rotating magnetic disk used to store and retrieve magnetically encoded data through a read-write head that is in close proximity to the surface of the disk.

**documentation**

Written descriptions, specifications, design, code, and comments (internal and external to a program) that make a program readable, understandable, and more easily modified; also, a user's manual for a program. *See* selfdocumenting code.

**doubly linked list**

Linked list in which each node is linked to both its successor and its predecessor.

**down**

A descriptive term applied to a computer when it is not in a usable condition.

**driver**

*See* test driver.

**dynamic allocation**

Creation of storage space in memory for a variable during run time. *See* referenced variable.

**dynamic data structure**

A data structure that may expand and contract during run time.

**dynamic storage**

*See* dynamic allocation.

**dynamic variable**

A variable created during execution of a program and hence not declared in the declaration section of a program.

**E****echo printing**

Printing the data values input to a program to verify that they are correct.

**edge**

Connection between two nodes in a graph; sometimes called arc.

**editor**

An interactive program that allows the user to create and alter text files such as data, programs, manuscripts, and so on.

**embedded computer system**

A computer system that is only a part of a larger piece of equipment.

**empty set**

The set with no members at all.

**encapsulation**

A programming language feature that allows a compiler to enforce information hiding.

**enumeration type**

An ordered set of literal values (identifiers) defined as a data type.

**error checking**

Explicit checking for invalid and error conditions in a program.

**exception handler**

A sequence of instructions that is executed when an exception is raised.

**executing**

The action of a computer performing as instructed by a given program.

**execution time**

*See* run time.

**explicit type conversion**

The conversion of one numeric type to another, accomplished by enclosing an expression in parentheses and preceding it with the name of the desired type.

**exponential notation**

*See* scientific notation.

**expression**

A sequence of identifiers and/or constants separated by compatible operators.

**external documentation**

Program specification, design and development history that are external to the body of executable code.

**external file**

A permanently stored file separate from the executing program.

**external pointer (to a list)**

A named access type variable that references the first node in a linked list.

**F****field identifier**

The name of a component in a record.

**field selector**

The expression used to access components of a record variable, consisting of the record variable name and the field identifier separated by a period.

**FIFO queue**

A data structure in which elements are added to the rear and removed from the front of the list; a "first-in, first-out" (FIFO) structure.

**file**

A named area in secondary storage that is used to hold a collection of data; the collection of data itself.

**file data type**

A data structure consisting of a sequence of components that are all of the same type; a collection of related data, usually stored on disk or tape, and referenced by a single name.

**file terminator**

A nonprintable control character or a sequence of nonprintable control characters that the system recognizes as marking the end of a file; not defined by the Ada language.

**finite state machine**

An idealized model of a simple computer consisting of a set of states, the rules that specify when states are changed, and a set of actions that are performed when changing states. *See* abstract data object.

**fixed-length string**

A string that contains a fixed (constant) number of characters.

**flag**

A Boolean variable that is set in one part of the program and tested in another to control the logical flow of a program.

**floating-point type**

A type used to approximate real numbers; uses a fixed number of digits (the mantissa) and a base raised to a power (the exponent).

**flow of control**

The order in which statements are executed in a program. *See* control structure.

**folding**

A hash method that breaks the key into several pieces and concatenates or exclusive-ORs some of them to form the hash address.

**for statement**

A looping control structure with predefined initial and final values for the loop control variable, as well as automatic incrementing of the loop control variable.

**formal parameter**

A parameter declared in a procedure specification, function specification, or generic declaration.

**formal parameter list**

The list of formal parameters contained in a procedure, function, or generic unit specification.

**formatting**

The planned positioning of statements or declarations and blanks on a line of a program; the arranging of program output so that it is neatly spaced and aligned.

**frame**

A section of code with which an exception handler is associated; most commonly block statements and subprograms.

**full binary tree**

Binary tree in which all the leaves are on the same level and every nonleaf node has two children.

**function**

A subprogram that is called from within an expression, and in which a single value (for example, the square root of a number) is computed and returned through the function name.

**functional decomposition**

A design methodology that works from an abstract functional description of a problem (top) to a detailed solution (bottom); a hierarchical approach to problem solving that divides a problem into functional subproblems represented by modules, which are easier to solve and which may themselves be further broken down into modules. The design consisting of a hierarchy of separate modules (solutions), with lower level modules containing greater detail than higher level modules.

**functional domain**

The set of valid input data for a program or subprogram.

**function call**

An expression requiring the computer to execute a function subprogram.

**function result**

The value computed by the function subprogram and then returned to the main program; often called just the result.

**functional cohesion**

A property of a module in which all concrete steps are directed toward solving just one problem, and any significant subproblems are written as abstract steps; the principle that a module should perform exactly one abstract action.

**functional equivalence**

A property of a module that it performs exactly the same operation as the abstract step it defines. A pair of modules are functionally equivalent to each other if they each accomplish the same abstract operation.

**G****general case**

In a recursive definition, the case for which the solution is expressed in terms of a smaller version of itself; recursive case.

**generic formal parameter**

A parameter defined in a generic unit declaration. Used to customize a generic unit for a specific problem.

**generic unit**

A template for a package or subprogram. Ada's generic units allow easy reuse of packages and subprograms.

**glass box testing**

*See* clear box testing.

**global identifier**

Any identifier declared outside of but accessible from a given declarative region.

**graph**

Data structure that consists of a set of nodes and a set of edges that relate the nodes to each other.

**graphic character**

A character that may be used as a character literal; a character other than a control character.

**H****hardware**

The physical components of a computer.

**hash function**

A function used to manipulate the key of an element in a list to identify its location in the list.

**hashing**

Technique used for ordering and accessing elements in a list in a relatively constant amount of time by manipulating the key to produce a (hopefully) unique location.

**header node**

Dummy node at the beginning of a list; used to simplify list processing.

**heap**

A complete binary tree, each of whose elements contains a value that is greater than or equal to the value of each of its children.

**height**

The number of levels in a tree.

**hierarchical records**

Records in which at least one of the fields is itself a record.

**high-level language**

A programming language that is closer to natural language than assembly language and whose statements each translate into more than one machine language instruction.

**homogeneous**

A descriptive term applied to structures in which all components are of the same data type (such as an array or file).

**homograph**

Each of multiple declarations with identical names. *See* name precedence.

**I****identifier**

A name associated with a process or object and used to refer to that process or object.

**implementing**

Coding and testing an algorithm.

**in mode**

A parameter passing mode; the value of the actual parameter is copied into the formal parameter when the subprogram is called; used when a subprogram requires a value.

**in out mode**

A parameter passing mode; the value of the actual parameter is copied into the formal parameter when the subprogram is called and the value of the formal parameter is copied back to the actual parameter when control is returned; used when a subprogram must change a value.

**in place**

Describes a kind of sorting algorithm in which the components in an array are sorted without the use of a second array.

**incomplete type declaration**

A type declaration that gives only the name of the type. It may include optional discriminants. An incomplete type may only be used in the declaration of access types.

**index**

A discrete value identifying a particular component of a data structure such as an array.

**infinite loop**

A loop whose terminating condition would never be reached; the loop would (theoretically) execute indefinitely and the program would never terminate.

**infinite recursion**

The situation in which a subprogram calls itself over and over without end.

**infix notation**

Notation for expressions in which the binary operator is placed between its operands.

**information hiding**

The practice of controlling access to the details of a module or structure.

**initialize**

To assign an initial value to a variable.

**inorder traversal**

Traversal of a binary tree in which each node is visited between its left and right subtrees.

**input**

Any external data used by a program, from whatever source, such as a keyboard or disk file.

**input/output (I/O)**

Media and devices used to achieve human/machine communication.

**input prompts**

Messages printed by an interactive program explaining what data is to be entered.

**insertion sort**

A sorting algorithm in which values are placed one at a time into their proper position within a list that was originally empty.

**instance**

A package or subprogram created from a generic unit.

**instantiation**

A declaration that creates an instance of a generic unit.

**integer number**

A positive or negative whole number made up of a sign and digits (when the sign is omitted, a positive sign is assumed).

**integration testing**

Testing that is performed on combined program modules that already have been independently tested.

**interactive processing**

Use of an interactive program; user interaction with a program, usually by prompts, data entry, and commands made through a terminal.

**interactive programming**

Use of an interactive system to create and compile programs through the use of an editor, compiler, debugger and other tools.

**interactive system**

Direct communication between the user and the computer, a terminal/computer connection allowing direct entry of programs and data and providing immediate feedback to the user.

**interface**

A connecting link (such as a computer terminal) at a shared boundary, allowing independent systems (such as the user and the computer) to meet and act on or communicate with each other; the formal definition of the function of a subprogram or package and the mechanism for communicating with it.

**internal documentation**

Features within the executable code that make a program easy to read and understand; includes comments, prettyprinting, and self-documenting code.

**interpreter**

A program that inputs a program in a high-level language and directs the computer to perform the actions specified in each statement; unlike a compiler, an interpreter does not produce a machine language version of the program.

**invoke**

*See* call.

**ISO**

International Organization for Standardization, an organization that promotes voluntary standards.

**iteration**

An individual pass through, or repetition of, the body of a loop.

**iterator**

An operation that allows us to process all of the components of a collection class object.

**K****key**

Field(s) in a record whose value(s) are used to determine the logical (and/or physical) order of the records in a list.

**key-ordered list**

A list in which the elements are ordered according to the value of a key field of each element.

**L****language independence**

A feature of abstract data types and algorithms that allows them to be implemented in almost any general-purpose programming language.

**leaf**

Node in a tree that has no children.

**left child**

Node to the left of a given node in a binary tree.

**left subtree**

All the nodes to the left of a given node in a binary tree.

**length**

The actual number of values stored in an array, list, or string.

**level**

The distance of a node from the root node.



**library unit**

A package declaration, subprogram declaration, package instantiation, or subprogram body with no corresponding declaration.

**linear probing**

Resolving a hash collision by sequentially searching a hash table beginning at the hash address.

**line terminator**

A nonprintable control character or a sequence of nonprintable control characters that the system recognizes as marking the end of a line of data; not defined by the Ada language.

**link time**

The time when compiled program units are combined into an executable program.

**linked list**

List in which the order of the elements is determined by an explicit link field in each element rather than sequential order in memory.

**linker**

A system program that combines compiled program units into an executable program.

**listing**

*See* source listing.

**literal**

Any value written directly in a program.

**local identifier**

An identifier declared in the block where it is used. *See* name precedence.

**local variable**

Variable declared within a subprogram and accessible only within the block in which it was declared; the value of this variable is destroyed when the procedure returns control to the calling program.

**logarithmic order**

For an algorithm, when the Big-O work expression can be expressed in terms of the logarithm of  $N$ , where  $N$  is the number of values in a data set.

**logical operator**

An operation applied to values of the type Boolean; in Ada these are the special symbols `and`, `or`, `xor`, `not`, `and then`, and `or else`.

**loop**

A control structure that allows a statement(s) to be executed more than once (until a termination condition is reached).

**loop control variable**

A variable (usually discrete) used to control the number of times the body of a loop is executed.

**loop invariant**

An assertion of what conditions must be true on entry into an iteration of a loop body and on exit from the loop.

**M****machine language**

The language used directly by the computer and composed of binary coded instructions.

**main storage**

Also main memory *See* memory.

**mainframe**

A large computing system designed for highvolume processing or for use by many people at once.

**maintenance**

The modification of a program, after it has been completed, in order to meet changing requirements or to take care of any errors that show up.

**mantissa**

With respect to floating-point numbers, the digits representing a number itself and not its exponent.

**maximum-height tree**

A tree whose height is equal to the number of nodes it contains.

**Max\_Int**

The predefined identifier in package System whose value is the largest (most positive) integer that can be represented in a given computer.

**membership test**

An operation that determines whether or not a value belongs to the domain of a range or subtype.

**memory**

The ordered sequence of storage cells (locations, words, places) in a computer that are accessed by address and used to temporarily hold the instructions and variables of an executing program. *See* secondary storage.

**memory unit**

The internal data storage of a computer. *See* memory.

**metric-based testing**

Testing in which goals are based on certain measurable factors.

**minimum-height tree**

A tree whose nodes are arranged so that the tree height is the minimum possible.

**Min\_Int**

The predefined identifier in package System whose value is the smallest (most negative) integer that can be represented in a given computer.

**model interval**

The interval defined by two adjacent model numbers.

**model numbers**

Real numbers that can be represented (stored) exactly.

**modular decomposition**

*See* functional decomposition.

**modular design**

A design methodology that breaks the program down into independent logical units or modules.

**module**

An independent unit that is part of a whole; a logical part of a design or program, such as a procedure.

**multi-dimensional array**

An array that requires multiple indices to select a component.

**N****name precedence**

When a homograph exists, the local identifier in a procedure takes precedence over any global identifier in any references that the procedure makes to that identifier.

**named association**

An association made between the actual and formal parameters by specifying the name of the formal parameter, an arrow symbol, `=>`, and the name of the actual parameter; also used in aggregates.

**named constant**

A location in memory, referenced by an identifier, where a data value that cannot be changed is stored.

**named number**

A named constant of type `universal_integer` or `universal_real`.

**named type**

A type defined in the declarative part of a package, procedure, or function.

**nested logic**

A control structure contained within another control structure.

**node**

A record containing data and one or more access values that designates other nodes. An element in a linked list or tree.

**null**

A constant in Ada that can be assigned to an access type variable, indicating that the variable designates (points to) nothing.

**null pointer**

An access value that designates (points to) nothing.

**null range**

A range that contains no values, the beginning value of the range is greater than the ending value.

**null string**

A string containing no characters.

## O

### object

In Ada, a value stored at some location in memory; variables and named constants, in object-oriented design, an entity that is a natural part of the problem and has a state and defined set of operations.

### object class (class)

The description of a group of objects with similar properties and behaviors. A template for creating individual objects.

### object code

The machine code produced by a compiler or assembler from a source program. Also called object program.

### object-oriented design (OOD)

A program design methodology based on defining classes of objects (entities that make sense in the context of the problem being solved).

### object-oriented programming (OOP)

A programming methodology that uses the properties of encapsulation, inheritance, and polymorphism to build and manipulate data types.

### observer

An operation that allows us to observe the state of an object without changing it.

### obsolete unit

A compilation unit that has not been recompiled after a unit on which it depends has been recompiled; an error is issued if an attempt is made to use an obsolete unit to form an executable program.

### one-dimensional array

A structured collection of components of the same type given a single name; each component is accessed by an index that indicates its position within the collection.

### operating system

The set of programs that manage computer resources.

### operator

A symbol that indicates an operation to be performed.

### operator precedence

*See* precedence rules.

### order of magnitude

Ways of expressing relationships between large numbers by using formal approximation; used in computing to express amount of work done.

### ordinal type

*See* discrete data type.

### out mode

A parameter passing mode; the value of the formal parameter is copied into the actual parameter when control is returned; used when a subprogram returns a result.

### output

Data produced by a program and sent to an external file or device.

### overflow

A condition where the results of a calculation are too large to represent on a given machine.

### overloading

The ability of an identifier to have several different meanings; subprogram names and enumeration literals are commonly overloaded.

## P

### package

A group of logically related entities that may include types and subtypes, objects of those types and subtypes, and subprograms with parameters of those types and subtypes; Ada's principle means of encapsulation.

### package body

The implementation of a package; its contents cannot be accessed by other program units.

### package declaration

The interface of a package. Describes the resources (constants, types, and operations) that may be used by other program units. It may include a private part that defines resources available only to the package body.

### palindrome

String that reads the same backward or forward (for example, RADAR).

### parameter declaration

The code that associates a formal parameter identifier with a data type and a passing mode.

### parameter list

A mechanism for communicating with a subprogram, via which data may be given to the subprogram and/or results received from it. *See* actual parameter list and formal parameter list.

### parent

The predecessor of a node in a binary tree. Each node except the root node has one parent.

### parent unit

A program unit containing the declaration of a subunit.

### path

Sequence of vertices that connects two nodes in a graph. Also, a combination of branches that might be traveled when a program is executed.

### path testing

A testing technique whereby the programmer tries to execute all possible paths in a program or subprogram.

### peripheral device

An input, output, or auxiliary storage device of a computer.

### pointer

*See* access type.

### pointer type variable

*See* access type variable.

### portability

The ability of software written for one computer to run successfully on different machines.

### positional association

A method of matching actual and formal parameters by their relative positions in the two parameter lists, also used in aggregates.

### postconditions

Assertions that state what results are to be expected at the exit of an operation or procedure, assuming that the preconditions are true.

### postfix notation

Notation for expressions in which the binary operator follows its operands.

### postorder traversal

Traversal of a binary tree in which each node is visited after its left and right subtrees.

### powerset

*See* universal set.

### pragma

A statement that directs the Ada compiler to perform compilation in certain ways.

### precedence rules

Rules that establish the order of importance of operators in a programming language.

### precision

The maximum number of significant digits in a float type.

### preconditions

Assertions that must be true on entry into an operation or procedure for the postconditions to be guaranteed.

### prefix notation

Notation for expressions in which the binary operator precedes its operands.

### preorder traversal

Traversal of a binary tree in which each node is visited before its right and left subtrees.

### prettyprinting

Program formatting to make a program more readable.

### priority queue

A data structure in which only the highest priority element can be accessed.



**private part**

The part of a package declaration containing declarations that are not available outside of the package.

**private type**

A type declared in the visible part of a package declaration whose details (record fields, array index, and component types, etc.) are declared in the private part of the package declaration; the type may be used outside of the package, but its internal components may not.

**procedure**

A subroutine that is executed when called.

**procedure call**

A statement that transfers control to a procedure, in Ada, this statement is the name of the procedure followed by a list of parameters.

**programming**

The planning, scheduling, or performing of a task or an event. *See* computer programming.

**programming in the large**

The design and implementation of solutions to problems at a scale too large to be carried out by a single programmer or even a small group of programmers.

**programming in the small**

The design and implementation of solutions to problems at a scale small enough to be carried out by a single programmer or a small group of programmers.

**programming language**

A set of rules, symbols, and special words used to construct a program.

**program unit**

A procedure, function, package, or generic unit.

**program verification**

The process of determining the degree to which a software product fulfills its specifications.

**propagated error**

Error in the result of an operation that is due to previous errors contained in the operands.

**propagated exception**

An exception that is raised again when control leaves a frame.

**pseudo-code**

A mixture of English and Ada-like control structures used to specify a design.

**Q****quadratic probing**

Resolving a collision by applying a rehash function of the form  $(\text{Hash\_Address} + I^2) \bmod \text{Array\_Size}$ , where  $I$  is the number of times that the rehash function has been applied.

**qualification**

Prefixing an identifier with the name of the package in which it is declared.

**qualified expression**

An expression preceded by an indication of its type or subtype; used when an expression's type is ambiguous.

**queue**

*See* FIFO queue or priority queue.

**queue overflow**

The condition resulting from trying to enqueue an element onto a full queue.

**queue underflow**

The condition resulting from trying to dequeue an empty queue.

**queuing system**

System made up of servers and queue(s) of objects to be served.

**R****raise an exception**

To abandon normal program execution, done when the system or program detects an abnormal condition.

**random access**

The process of retrieving or storing elements in a data structure where the time required for such access is independent of the arrangement of the elements.

**random probing**

Resolving a collision by reaching random locations in a hash table for the hash target.

**range**

The set of values between a specified first and last value, including those values.

**range-checking**

The automatic detection of an out-of-range value being assigned to a variable.

**reading marker**

An indicator of the next component to be read.

**real number**

A number that has a whole and a fractional part and no imaginary part; in Ada, approximated by floating-point or fixed-point types.

**record**

A composite data type with a fixed number of components that are accessed by name, not by an index; the components may be of different types.

**recursion**

The ability of a procedure or function to call itself.

**recursive algorithm**

A solution that is expressed in terms of (a) smaller instances of itself and (b) a base case.

**recursive call**

A subprogram call in which the subprogram being called is the same as the one making the call.

**recursive case**

*See* general case.

**recursive definition**

A definition in which something is defined in terms of smaller versions of itself.

**referenced variable**

A variable accessed not by name but through an access variable; a dynamic variable; a variable created by Ada's `new` operator.

**refinement**

In functional decomposition, the expansion of a module specification to form a new module that solves a major step in the computer solution of a problem.

**regression testing**

Re-execution of program tests after modifications have been made in order to ensure that the program still works correctly.

**rehashing**

Resolving a collision by computing a new hash location from a hash function that manipulates the original location rather than the element's key.

**relational operator**

An operator that states that a relationship exists between two values; in Ada, a symbol that causes the computer to perform operations to verify whether or not the indicated relationship exists.

**relative address**

The offset (number of memory locations) from some other address determined at run time.

**representational error**

An arithmetic error that occurs when the precision of the result of an arithmetic operation is greater than the precision of a given machine.

**requirements**

Statement of what is to be provided by a computer system or software product.

**reserved word**

An identifier that has a specific meaning in a programming language and may not be used for any other purpose in a program.

**return**

The point at which execution of a subprogram is completed and execution resumes with the statement immediately following the call.

**right child**

Node to the right of a given node in a binary tree.

**right-justified**

Placed as far to the right as possible.

**right subtree**

All the nodes to the right of a given node in a binary tree.

**robustness**

The ability of a program to recover following an error and to continue operating within its environment.

**root**

The top, or base, node of a tree structure. The external pointer to a tree data structure.

**round off**

To truncate (or make zero) one or more least significant digits of a number, and to increase the remaining least significant digit by one if the truncated value is more than half of the number base. In Ada, an explicit conversion from a float type to an integer type rounds off the real value to the nearest integer.

**rounding error**

The error in the approximation of a real number by a model number.

**row-major order**

An array organization where the elements are stored in memory row by row.

**run time**

The phase of program execution during which program instructions are performed.

**run-time error**

Software error that occurs during the execution of the program.

**run-time stack**

A data structure that keeps track of activation records during the execution of a program.

**S****scalar data type**

A data type in which the values are ordered and each value is atomic (indivisible).

**scientific notation**

A method of representing a number as an expression consisting of a number between 1 and 10 multiplied by the appropriate power of 10. Also called floating-point notation.

**scope**

*See* scope of access.

**scope of access**

All of the places from which an identifier can be accessed; often referred to simply as its scope.

**scope rules**

The rules that determine where in a program a given identifier may be accessed.

**searching**

The locating of a particular element in a data structure.

**secondary storage**

Backup storage for the main storage (memory) of a computer usually permanent in nature (such as tape or disk).

**seed**

Variable that initializes a random number generator.

**selection**

A control structure that selects one of possibly several options or paths in the flow of control, based upon the value of some expression.

**self-documenting code**

Source code that uses meaningful identifier names, as well as effective use of clarifying comments.

**semantics**

The set of rules that gives the meaning of instructions written in a programming language.

**sentinel**

A special data value used to mark the end of a data file.

**sequential access**

The process of retrieving or storing elements in a fixed order in a data structure where the time required for such access is dependent on the order of the elements.

**sequence**

A structure in which statements are executed one after another.

**sequential file**

A binary file whose components can be accessed sequentially.

**sequential search**

A search technique in which elements are examined sequentially until the target element is found.

**set**

A class composed of an unordered collection of distinct elements (members) chosen from the values of the base type.

**short-circuit operators**

A logical operator (either `or else` or `and then`) that stops the evaluation of additional terms in a Boolean expression as soon as the result is known.

**siblings**

Nodes in a tree that have the same parent node.

**side effect**

Any effect of one module on another module that is not a part of the explicitly defined interface between them.

**SIGAda**

An ACM special interest group for using the Ada language to develop software.

**significant digits**

Those digits from the first nonzero digit on the left to the last nonzero digit on the right (plus any zero digits that are exact).

**simple type**

*See* atomic data type.

**simulation**

A problem solution that has been arrived at through the application of an algorithm designed to model the behavior of physical systems, materials, or processes.

**size (of an array)**

The physical space reserved for an array.

**skewed tree**

A subjective term describing the shape of a binary tree. The height of a skewed tree is closer to the maximum-height tree than the minimum-height tree.

**slice**

A contiguous portion of a one-dimensional array indicated by an index range; frequently used with strings.

**software**

Computer programs; the set of all programs available to a computer.

**software engineering**

A disciplined approach to the design, production, and maintenance of computer programs that are developed on time and within cost estimates, utilizing tools that help to manage the size and complexity of the resulting software products.

**software specifications**

Detailed description of the function, inputs, processing, outputs, and special requirements of a software product, which provides the information needed to design and implement the program.

**sorted (value-ordered) list**

A list in which the elements are ordered according to their value or the value of a key field of each element.

**sorting**

Arrangement of elements in a list according to the increasing (or decreasing) values of some key field of each element.

**source code**

Also called source program; a program in its original form, in the language in which it was written, prior to any compilation or translation.

**source listing**

A printout of a source program processed by a compiler and showing compiler messages, including any syntax errors in the program. Also called compilation listing.

**specifications**

The formal definition of a problem to be solved, describing its inputs and outputs and pre- and postconditions.

**stack**

A data structure in which elements are added and removed from only one end. a "last in, first out" (LIFO) structure.

**stack overflow**

The condition resulting from trying to push an element onto a full stack.

**stack underflow**

The condition resulting from trying to pop an empty stack.

**statement**

An instruction in a programming language.

**static allocation**

Creation of storage space in memory for a variable at compile or link time (amount of space cannot be changed at run time).

**static data structure**

A data structure fixed in size at compile time. *See* static allocation.

**step-wise refinement**

A design method in which an algorithm is specified at an abstract level and additional levels of detail are added in successive iterations throughout the design process. *See* functional decomposition.

**storage**

*See* memory.

**string**

A collection of characters interpreted as a single data. *See* fixed-length string, varying-length string, bounded-length string, and unbounded-length string,

**structured design**

A design methodology incorporating a high degree of modularity, and employing generic control structures having only one entry and one exit. *See* functional decomposition.

**structured programming**

The use of structured design and the coding of a program that parallels the structure of the design.

**structured type**

*See* composite data type.

**stub**

A dummy procedure or function that assists in testing part of a program; it has the same name and interface as a procedure or function that would actually be called by the part of the program being tested, but it is usually much simpler.

**subprogram**

A procedure or function.

**subprogram body**

The implementation of a subprogram; its specification can act as the subprogram declaration if a declaration is not already in the program library.

**subprogram declaration**

A specification of a subprogram; commonly written in a package declaration to specify operations provided by the package.

**subroutine**

*See* subprogram.

**subscript**

*See* index.

**subscripted variable**

*See* array.

**subset**

The set A is a subset of the set B if each element of A is an element of B.

**subtree**

A portion of a tree. Each node in a tree is a root of a subtree.

**subtype**

A subset of a type determined by a constraint on the type.

**subunit**

A separately compiled body that is declared in the declarative part of another program unit.

**symbol table**

Table, defined by the compiler, that maps variables to memory locations.

**syntax**

The formal rules governing the construction of valid statements in a language.

**syntax diagram**

A pictorial definition of the syntax rules of a programming language.

**syntax error**

Invalid construction of statements in a programming language, which is detected during the compilation of the program.

**system software**

The set of programs that improves the efficiency and convenience of using a computer, such as the operating system, editor, and compiler.

**T****tail recursion**

A recursive algorithm in which no statements are executed after the return from the recursive call; often indicates the problem could be solved more efficiently with iteration.

**tape**

A secondary mass storage medium providing a large amount of permanent storage; a thin plastic strip having a magnetic surface used to store and retrieve magnetically encoded data through a read/write head that is in close proximity to the surface of the tape.

**team programming**

The use of two or more programmers to design a program that would take one programmer too long to complete; the type of programming for which Ada was designed.

**test driver**

A simple dummy program that sets up the testing environment by declaring and assigning initial values to variables, and then calls the subprogram to be tested.

**test plan**

A document showing the test cases planned for a program or module, their purposes, inputs, expected outputs, and criteria for success.

**testing**

Checking a program's output by comparing it to hand-calculated results; running a program with data sets designed to discover any errors.

**text file**

A file data type whose components are characters organized as a collection of lines; a line terminator follows every line; a file terminator follows the last line in the file.

**time sharing**

A method of operation in which a computer is shared by several users simultaneously.

**top-down design**

*See* functional decomposition.

**top-down programming**

Programming that incorporates functional decomposition, and, through the use of procedures, functions, and control structures, maintains in the program the modularity and structure of the design.

**top-down testing**

A technique for testing the modules (procedures and functions) of a program, as they are written, by calling them with actual parameters and providing stub (dummy) modules for those modules not yet written but referenced in the program.

**trace**

To follow the logical flow of a program and determine the value of variables after each instruction. Also known as code walk-through and playing computer.

**trailer node**

Dummy node at the end of a list; used to simplify list processing. The key of a trailer node must be greater than any valid key for the list.

**translator**

A program that translates from one programming language to another (usually machine code). *See* assembler, compiler, interpreter.

**tree**

A data structure composed of a root node having off-spring that are also nodes that can have offspring, and so on.

**tree diagram**

A hierarchical chart showing the relationships of modules in a top-down design.

**truncation**

The decrease in precision of a number by the loss or removal of one or more least significant digits.

**two-dimensional array**

A collection of components, all of the same type, structured in two dimensions; each component is accessed by a pair of indices that represent the component's position within each dimension.

**type**

*See* data type.

**type cast**

*See* explicit type conversion.

**type coercion**

An automatic assignment of a value of one type to a variable of another type; in Ada, available only with universal types.

**type definition**

The association of a type identifier with the definition of a new type or subtype in a declarative region.

**U**

**unary operator**

An operator requiring only one operand such as the logical operator `not`.

**unbounded-length string**

A varying-length string with no bound on the number of characters.

**unconstrained array type**

An array type without a range specified for its index; only the index type and component type are specified.

**underflow**

A condition that occurs when the results of a calculation are too small to represent in a given machine.

**undirected graph**

A graph in which the edges have no direction.

**unit testing**

Testing that is performed to verify that a single unit (procedure, module, or package) of a program meets its specifications.

**universal set**

The set consisting of all values of the base type.

**universal types**

The types of numeric literals; integer literals are type `universal_integer`, and float literals are type `universal_real`.

**V**

**validated Ada**

An Ada implementation that has successfully passed the testing procedures of the Ada Validation Office and received a certificate of validation.

**variable**

A location in memory, referenced by an identifier, in which a data value that can be changed is stored.

**variable declaration**

The creation of a variable in Ada in the declaration region of a block with the variable identifier on the left of the colon (`:`) and the type identifier on the right.

**varying-length string**

A string in which the number of characters can change.

**visible part**

The part of a package declaration containing declarations that can be used outside of the package.

**verification**

The process of determining the degree to which a software product fulfills its specifications.

**vertex (pl. vertices)**

A node in a graph.

**W**

**weighted graph**

Graph in which each edge carries a value.

**white box testing**

*See* clear box testing.

**word**

A group of bits, one or more bytes in length, treated as a unit or location in memory, and capable of being addressed.

**word size**

The number of bits comprising a word or location in memory.

**work**

A measure of the effort expended by the computer in performing a computation. *See* order of magnitude.

## Answers to Selected Exercises

### Chapter 1

1. Software engineering is a disciplined approach to the creation and maintenance of computer programs throughout their whole life cycle.
4. Some software tools used in developing computer programs are text editors, compilers, assemblers, operating systems, and debugging programs.
7. Software specifications tell exactly what a program will do, but not how it will do it.
9. Ways to make your programs readable and easily understood include modular programming; use of comments, self-documenting code, prettyprinting, and named constants; and avoidance of "tricky" code.
12. The first step in developing any software program is to think.
13. When requirements in a programming assignment are ambiguous, you should ask the instructor to clarify them. If that isn't possible (for instance, if it's 2:30 Sunday morning), you should make your best guess as to what is required and explicitly document your interpretation in the program's Assumptions section.
18. Information hiding is the practice of controlling access to the details of a module or structure. The details necessary to display a floating-point number are hidden within the `Put` procedure in package `Float_IO`.
22. Object classes have attributes and operations associated with them.
25. *Implementation* refers to the actual solving of the problem in a way that can be executed on a computer. Implementation answers the *how* questions.
27. To separate the specification of a package from its implementation.
29. (a) Examples of  $O(1)$  algorithms: printing a character to the screen, incrementing a variable, adding two numbers together.  
 (b) Examples of  $O(N)$  algorithms: initializing all of the elements in a one-dimensional array to zero, incrementing all the elements in a one-dimensional array, multiplying two numbers by performing successive addition operations, raising a number to a power by performing successive multiplication operations.  
 (c) Examples of  $O(N^2)$  algorithms: initializing all of the elements in a two-dimensional array to zero, printing out all the elements in a two-dimensional array, searching for the smallest element in an unsorted two-dimensional array.
32. Algorithm 3 [ $O(\log_2 N)$ ] should execute the fastest; Algorithm 2 [ $O(N)$ ] should execute the slowest.
34. *This Week*

|                 |                                                                          |
|-----------------|--------------------------------------------------------------------------|
| Monday          | THINK.                                                                   |
| Tuesday         | Write detailed description. Get answers to questions about requirements. |
| Wednesday       | Write top-down design level 0.                                           |
| Thursday-Sunday | Write detailed levels of top-down design.                                |

## Next Week

|                  |                                                   |
|------------------|---------------------------------------------------|
| Monday           | Begin coding from detailed design.                |
| Tuesday          | Complete coding (program compiles without error). |
| Wednesday-Friday | Test program. (Finish before 2:00 PM Friday.)     |
| Weekend          | Attend sister's wedding.                          |

Two Weeks from  
Now

|         |                                                                            |
|---------|----------------------------------------------------------------------------|
| Monday  | Double-check that all documentation is ready to be turned in with program. |
| Tuesday | Turn in program.                                                           |

**Chapter 2**

1. (a) False; software verification begins at the start of the software lifecycle.  
(b) False; testing is just one part of the program verification effort.  
(c) False; testing does not "prove" program correctness.  
(d) True.  
(e) False; syntax errors are usually detected by the compiler.
4. A syntax error is an error in the use of the programming language; it's most likely found at compile time, if it has not been detected by visual inspection of the program. A logical error is an error in how the program works; it's most likely detected at run time when it causes the program to crash or to produce wrong results.
7. Some examples of run-time errors occurring as the result of a programmer making too many assumptions: attempting to divide without checking the value of the denominator (to avoid dividing by 0), attempting to use data read from user inputs without checking its value, attempting to use the value in a data structure without checking to see if the structure is empty.
9. Designing for correctness involves using program design strategies in an attempt to ensure that the program will be correct. The types of assertions described in this chapter that can contribute to the correctness of a design are preconditions/postconditions and loop invariants.
12. (a) A loop invariant is an assertion of the conditions that must be true on entry into an iteration of the loop body and that are true on exit from the loop.  
(b) Loop Invariant:  $1 \leq \text{Index} \leq \text{Element\_Count} + 1$  AND Target\_Value does not exist in Data\_List(1..Index - 1).
15. We must always check that the loop will terminate.
18. The purpose of conducting design and code inspections is to identify errors in programs as early as possible, before the actual execution of the program during testing.
22. A single programmer could use the inspection process as a way to do a structured "desk check" of his or her program and would especially benefit from inspection checklists of errors to look for.
24. Unit testing is the testing of a single unit of the program (for instance, a procedure or function). Integration testing is the testing of groups of already tested units to make sure that they interact correctly and that the whole program works according to its specifications.
27. Data coverage refers to testing the possible inputs or classes of inputs that may occur in the execution of the program. Code coverage refers to testing the structure of the program's solution by driving the execution into the various paths or branches of the program. Neither strategy is necessarily better than the other. The best type of testing to use depends on the testing strategy and requirements.
31. Branches and paths refer to code segments (and combinations of segments) that should be executed in various test cases. It is possible to attempt 100% *branch* coverage of a program; this means that every branch in the program will be executed in some test case. It is not possible to

attempt 100% path coverage of anything but the most trivial programs because of the explosion of test cases that would have to be generated.

36. (a) The Ada compiler ensures that `Value` can only contain a whole number.
- (b) Ada automatic range checking at run time ensures that `Chicken` is between 1 and 12.

Chapter 3

3. The composite types predefined in Ada are arrays, records, and files.
4. The accessing function of a one-dimensional array is direct access through the element's index.

6. (a) 0 (k) A
- (b) 4 (l) F
- (c) 5 (m) 5
- (d) 0..4 (n) A..F
- (e) True (o) False
- (f) (True, True, True, True, False) (p) True
- (g) (True, True, True, True, True) (q) (24, 35, 40, 32, 19)
- (h) False (r) (18, 33, 41, 35, 24)
- (i) False (s) True
- (j) False

8. (a) `type Weather_List_Type is array (Month_Type) of Weather_Rec;`  
`Yearly_Weather : Weather_List_Type;`
- (b) `Yearly_Weather(July).Actual_Rain := 1.05;`
- (c) 238

| 11. | <i>Field</i>  | <i>Length</i> | <i>Offset</i> |
|-----|---------------|---------------|---------------|
|     | First_Name    | 10            | 0             |
|     | Last_Name     | 10            | 10            |
|     | ID            | 1             | 20            |
|     | GPA           | 2             | 21            |
|     | Current_Hours | 1             | 23            |
|     | Total_Hours   | 1             | 24            |

13. 2,500 cells

17. (a) The object is encapsulated within the package body. No other program unit can access resources defined within a package body.
- (b) By declaring the details in the private part of the package specification.
- (c) The application programmer can declare many objects of a given class.
- (d) The object is fully encapsulated. The user of the package is unable to even see the details of the object (the details in a private type can be seen but not accessed).

20. (a) `My_Card : Bingo_Card_ADT.Card_Type;`  
`Your_Card : Bingo_Card_ADT.Card_Type;`
- (b)

`type Card_Array is array (Positive range <>) of Bingo_Card_ADT.Card_Type;`

- (c)
- ```
function Num_Winners (Cards : in Card_Array) return Natural;  
  
-- Returns the number of Bingo Cards that are winners. (A winner  
-- is a card in which all the numbers in a row, column or diagonal of  
-- the card are marked.)  
--
```

```

-- Preconditions : All Bingo cards in the array are valid
--                (successfully read with Get Card).
--
-- Postconditions : The number of winning cards is returned.

(d) function Num_Winners (Cards : in Card_Array) return Natural is
    Result : Natural ;
begin
    Result := 0;
    for Index in Cards'Range loop
        if Bingo_Card_ADT.Winner (Card => Cards(Index)) then
            Result := Result + 1;
        end if;
    end loop;
    return Result ;
end Num_Winners;

```

Chapter 4

2. (a)

```

type Month_Type is (January, February, March, April, May, June, July,
                    August, September, October, November, December);

```

```

package Month_Set is new Discrete_Set (Element_Type => Month_Type);

```

```

(b) Winter_Months : Month_Set.Set_Type;
    Fall_Months    : Month_Set.Set_Type;
    Cool_Months     : Month_Set.Set_Type;

```

```

(c)

```

```

Winter_Months := Month_Set.Empty_Set + December + January + February;

```

```

(d) Fall_Months := Month_Set.Empty_Set;
    for Month in September..November loop
        Fall_Months := Fall_Months + Month;
    end loop;

```

```

(e) Cool_Months := Fall_Months + Winter_Months;

```

```

5. procedure Put_Set(Set : in Set_Type) is
begin
    for Index in Element_Type loop
        if Set (Index) then
            Text_IO.Put_Line (Element_Type'Image(Index));
        end if;
    end loop;
end Put_Set;

```

6. When the variable declaration for Nums is elaborated, the system attempts to find enough memory for an array dimensioned from Min_Int to Max_Int. If the computer does not have enough memory for such a large array, STORAGE_ERROR is raised.

```

10. (a) generic
    type Value_Type is private;
    procedure G_Swap (First_Value : in out Value_Type;
                     Second_Value : in out Value_Type);
(b) procedure Swap is new G_Swap (Value_Type => Integer);
    -- A procedure that swaps the values of two integer variables

```



```

(c) procedure G_Swap (First_Value : in out Value_Type;
                     Second_Value : in out Value_Type) is
    Temp : Value_Type;
begin
    Temp      := First_Value;
    First_Value := Second_Value;
    Second_Value := Temp;
end G_Swap;

```

```

12. (a) generic
    type Value_Type is (<>);
    type Array_Type is array (Integer range <>) of Value_Type;
    procedure Maximum (List : in Array_Type;
                      Largest : out Value_Type);
(b) type My_Array_Type is array (Integer range <>) of Natural;
    procedure Max is new Maximum (Value_Type => Natural;
                                  Array_Type => My_Array_Type);
(c) procedure Maximum (List : in Array_Type;
                      Largest : out Value_Type) is
    Max_So_Far : Value_Type;
begin
    Max_So_Far := List (List'First);
    for Index in List'First + 1 .. List'Last loop
        if List(Index) > Max_So_Far then
            Max_So_Far := List(Index);
        end if;
    end loop;
    Largest := Max_So_Far;
end Maximum;
(d) generic
    type Value_Type is private;
    type Array_Type is array (Integer range <>) of Value_Type;
    with function ">" (Left : in Value_Type;
                      Right : in Value_Type) return Boolean;
    procedure Maximum (List : in Array_Type;
                      Largest : out Value_Type);
(e) type My_Array_Type is array (Integer range <>) of Natural;
    procedure Max is new Maximum (Value_Type => Natural;
                                  Array_Type => My_Array_Type;
                                  ">"      => Standard.">");

```

(f) No changes are necessary.

Chapter 5

3.

```

function To_String40 (Value : in String132.Bounded_String)
                    return String40.Bounded_String is
    Result : String40.Bounded_String;
begin
    if String132.Length (Value) > 40 then
        Result := String40.To_Bounded_String
            (String132.To_String (String132.Slice (Value, 1, 40)));
    else
        Result := String40.To_Bounded_String (String132.To_String (Value));
    end if;
    return Result;
end To_String40;

```

6. (a) 2 cells

(b) 2,500 cells

```
(c) Student_List.all (1).ID := 1000;
(d) for Index in 1..1000 loop
    Integer_IO.Put (Item => Student_List.all(Index).ID,
                   Width => 10);
    Text_IO.New_Line;
end loop;
```

(e)

```
for Index in 1..1000 loop
    Student_List.all(Index).Total_Hours :=
        Student_List.all(Index).Total_Hours +
        Student_List.all(Index).Current_Hours
    Student_List.all(Index).Current_Hours := 0;
end loop;
```

(f)

```
procedure Free is new Unchecked_Deallocation (Object => List_Type,
                                              Name    => List_Ptr);
(g) Free (Student_List);
    null
```

8. (a) No memory leak. No CONSTRAINT_ERROR.

(b) CONSTRAINT_ERROR.

(c) Memory leak.

(d) Memory leak and CONSTRAINT_ERROR (A is null).

(e) Memory leak and CONSTRAINT_ERROR ('f' not valid index for B.all).

(f) Memory leak.

(g) No memory leak. No CONSTRAINT_ERROR.

(h) CONSTRAINT_ERROR (index ranges of A.all and C.all differ).

(i) Memory leak.

10. (a) Limited private types prevent the application programmer from using the assignment operator with access values. The assignment operator is the most common source of aliases and memory leaks.

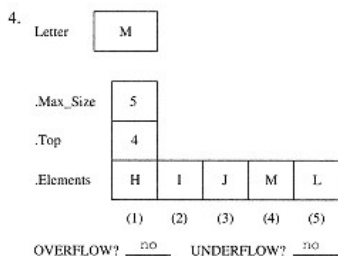
(b) The application programmer may still use the limited private access types to declare local objects in subprograms. Because local variables become undefined when control leaves a subprogram, any memory they designate is lost when control leaves the subprogram.

Chapter 6

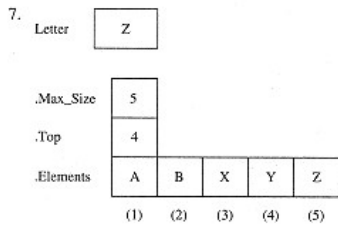
1. A stack is an ordered group of homogeneous elements; elements are added to and removed only from the top of the stack.

3. (a) package Name_Stack is new Stack (Element_Type => Name_Rec);
 (b) Names : Name_Stack.Stack_Type (Max_Size => 250);

4.



7.



9. The answer is (a).

11. (a) Cannot be done as a function. Because the stack is changed (temporarily) we must use `in out` mode to pass the stack. The following assumes that a `use` clause makes the stack operations directly visible.

```

procedure Top (Stack : in out Stack_Type;
              Value : out Element_Type) is
    Temp : Element_Type;
begin
    Pop (Stack, Temp);
    Push (Stack, Temp);
    Value := Temp;
end Top;
(b) function Top (Stack : in Stack_Type) return Element_Type is
begin
    if Stack.Top = 0 then
        raise UNDERFLOW;
    else
        return Stack.Elements (Stack.Top);
    end if;
end Top;

```

14. (a) This implementation violates the idea of an array as a homogeneous collection of data. Though the elements are the same type (Integer) as the index of the top element, they are different in terms of their purpose in the program. The use of the array is correct syntactically, but might be considered to be incorrect semantically.

(b) This implementation choice would make no change in the stack specifications. The procedures and function bodies would need to be modified slightly from the array-based implementation in the chapter, changing references to "Stack.Top" to "Stack(0)" and references to "Stack.Elements" to "Stack".

15. The last node does not point to another node; rather, its `Next` field contains a null value to indicate that this is the end of the stack.

17. (a) pointer (e) pointer
 (b) pointer (f) pointer
 (c) array (g) integer
 (d) integer

```

21. Clear (Temp_Stack);
    loop
        exit when Empty (Pez_Stack);
        Pop (Stack => Pez_Stack, Popped_Element => Candy);
        if Candy is Yellow then
            Eat (Candy);
        else
            Push (Stack => Temp_Stack, New_Element => Candy);
        end if;
    end loop;

    loop
        exit when Empty (Temp_Stack);
        Pop (Stack => Temp_Stack, Popped_Element => Candy);
        Push (Stack => Pez_Stack, New_Element => Candy);
    end loop;

```

23.	<i>Number of Elements</i>	<i>Static Array-Based</i>	<i>Dynamic Linked Stack</i>
	0	1204	4
	10	1204	164
	50	1204	804
	100	1204	1604

25. (a) 3

(b) 3

(c) 25

```

29. (a) procedure Copy (Target : in out Stack_Type;
                      Source : in Stack_Type) is
    begin
        Target := Source;
    end Copy;

```

(b) We must make a copy of the stack, not an alias.

```

procedure Copy (Target : in out Stack_Type;
                Source : in Stack_Type) is
    Source_Ptr : Stack_Type; -- Designates current node in Source
    Target_Ptr : Stack_Type; -- Designates last node in Target
begin
    Clear (Target);
    if Source /= null then -- Do nothing if Source is empty
        Target := new Node_Type'(Source.all); -- 1st node
        Source_Ptr := Source; -- Initialize traverse of source stack
        Target_Ptr := Target; -- Set to last node of Target
        loop -- Exit when Source_Ptr designates last node in Source
            exit when Source_Ptr.all.Next = null;
            -- Add a copy of the current Source node to the end of the Target
            Target_Ptr.all.Next := new Node_Type'(Source_Ptr.all.next.all);
            -- Advance both pointers to the next node
            Source_Ptr := Source_Ptr.all.Next;
            Target_Ptr := Target_Ptr.all.Next;
        end loop
    end if;
end Copy;

```

Chapter 7

1. A FIFO queue is an ordered group of homogeneous elements that satisfy the following property: New elements are added at the rear, and old elements are removed from the front.

4. (a) no (f) yes
 (b) yes (g) no
 (c) no (h) yes
 (d) no (i) no
 (e) yes

6.

Queue.Items				
V	W	X	Y	J
(1)	(2)	(3)	(4)	(5)

Queue.Front = 1
 Queue.Rear = 5
 Queue.Count = 5

OVERFLOW no UNDERFLOW no

8.

Queue.Items				
V	W	L	Y	Z
(1)	(2)	(3)	(4)	(5)

Queue.Front = 4
 Queue.Rear = 3
 Queue.Count = 5

OVERFLOW no UNDERFLOW no

10.

Queue.Items				
V	W	X	Y	Z
(1)	(2)	(3)	(4)	(5)

Queue.Front = 1
 Queue.Rear = 3
 Queue.Count = 3

OVERFLOW no UNDERFLOW no Value Z

```
14. function Empty (Queue : Queue_Type) return Boolean is
    -- The queue is empty if it is not full and
    -- the rear pointer is just behind the front pointer.
begin
    return not Full(Queue) and
           Queue.Rear rem Queue.Max_Size + 1 = Queue.Front;
end Empty;
```

```
15. procedure Dequeue (Queue : in out Queue_Type;
    Item : out Element_Type) is
begin
    if Empty (Queue) then
        raise UNDERFLOW;
```

```

else
  Item := Queue.Items(Queue.Front);
  Queue.Front := Queue.Front rem Queue.Max_Size + 1;
  Queue.Full := False;
end if;
end Dequeue;

```

20. Use exception handlers when queue overflow or underflow are not part of normal processing in the given problem. If these conditions are part of normal processing, use the `Full` and `Empty` functions. For example, the algorithm to print a manifest list for a train in the Hump Yard application dequeues cars until the queue is empty. An empty queue is expected in this processing and therefore the function `Empty` is used rather than the `UNDERFLOW` exception.

22. (a) The space for the queue elements is allocated dynamically. (The space for all possible elements is allocated at the same time.)

(b) The space for the variable `Queue` is the size of an access value.

```

(c) procedure Create_Queue (Queue : in out Queue_Type;
                           Max_Size : in Positive) is
begin
  Queue := new Queue_Data_Type (Max_Size => Max_Size);
end Create_Queue;

```

(d) `Full` is a meaningful operation; it indicates whether there is space available in the array.

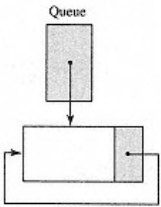
```

function Full (Queue : in Queue_Type) return Boolean is
begin
  return Queue.all.Count = Queue.all.Max_Size;
end Full;

```

(e) You should modify the source code for the array-based implementation. The only change to the code is to replace references to "`Queue`" with "`Queue.all`".

25. (a)



```

(c) function Empty (Queue : in Queue_Type) return Boolean is
begin
  return Queue = null;
end Empty;

```

```

(d) procedure Enqueue (Queue : in out Queue_Type;
                      Item : in Element_Type) is
begin
  if Queue = null then -- Is the queue empty?
    Queue := new Node_Type;

```

```

        Queue.all := (Info => Item,
                      Next => Queue);
    else
        Queue.all.Next := new Node_Type'(Info => Item,
                                           Next => Queue.all.Next);
        Queue := Queue.all.Next;
    end if;
end Enqueue;

27. procedure Add_Job (Job_Queuees : in out Job_Array;
                     ID           : in   ID_Type;
                     Token        : in   Token_Type) is
begin
    Token_Queue.Enqueue (Queue => Job_Queuees (ID / 100),
                        Item  => Token);
end Add_Job;

29. procedure Clean_Up_Jobs (Job_Queuees : in out Job_Array;
                           Message      : in   String) is
    Token : Token_Type;
begin
    -- Each iteration one job queue is cleared and its users notified
    for Index in Job_Queuees'Range loop
        loop -- Each iteration one job is cleared from current queue
            exit when Token_Queue.Empty (Job_Queuees(Index));
            Token_Queue.Dequeue (Queue => Job_Queuees(Index),
                               Item  => Token);
            Notify (Token, Message);
        end loop;
    end loop;
end Clean_Up_Jobs;

```

Chapter 8

1. (a) function Number_Of (Account : in Account_Rec) return Positive is
begin
return Account.Account_Number;
end Number_Of;

(b)

```

package Account_List is new Sorted_List (Element_Type => Account_Rec,
                                         Key_Type      => Positive,
                                         Key_Of        => Number_Of,
                                         "="          => Standard."=",
                                         "<"          => Standard."<");

```

(c)

```

procedure Put_One_Excellent (Account : in out Account_Rec) is
begin
    if Account.Credit_Rating = Excellent then
        Text_IO.Put_Line (Account.Company_Name);
    end if;
end Put_One_Excellent;

procedure Put_All_Excellent is new Account_List.Traverse
    (Process => Put_One_Excellent);

```

4. (a) In a sequential list, the successor of each element is implicit from the element's physical position in the list; elements are placed in sequential slots within the structure. In a linked representation, the order of the elements is determined by explicit links between them.

(b) Sequential lists are better solutions for problems in which random access to the elements is needed and in which the number of elements is known and stable. A sequential list might be employed, for example, for a list of student information for a class of 40 students, each with a student number from 1 to 40. Such a list might be used in meeting with individual students to discuss their progress and in generating report cards.

(c) Linked lists are better solutions for problems in which you cannot predict the number of elements or in which the number of elements varies widely. Linked lists are more appropriate for accessing elements one after another than for accessing elements randomly. A linked list might be employed, for example, for a list of employee records that is used to create phone lists and for producing paychecks.

7. (a) 30

(b) 90

(c) 45

10.

```
(a) List := A.all.Next;
(b) B := B.all.Next;
(c) List := null; -- Memory leak
(d) A.all.Next.all.Info := 46;
```

13. 4

3

2

1

0

16. 2

4

6

19. (a)

generic

```
-- The user must supply a procedure that will be called once for every
-- element in the list. Process may not change the key of Element.
with procedure Process (Element : in out Element_Type);
```

```
procedure Traverse (List : in out List_Type);
-- Calls procedure Process once for every element in the list. Elements
-- are "processed" from smallest key to largest key.
```

```
-- Preconditions : None
--
-- Postconditions : For every element in List, Process(Element)
--
-- Exceptions : KEY_ERROR If Process attempts to change the Key
-- of any element. Any elements modified
-- before the exception remain modified.
-- Elements not processed are not modified.
```

(b) Traverse could create a copy of the list and then process all of the elements in the copy. If no exception is raised, replace the original list with the possibly modified copy.

22. (a)

```

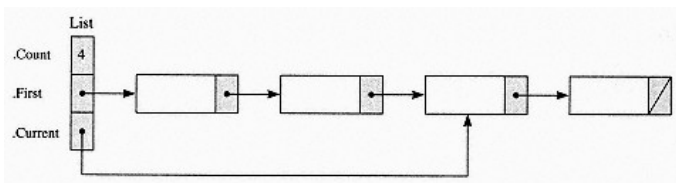
procedure Merge_Lists (List_1   : in out List_Type;
                      List_2   : in out List_Type;
                      New_List  : out List_Type);
  Index_1 : Positive;
  Index_2 : Positive;
  Index_New : Positive;
begin
  Index_1 := 1;
  Index_2 := 1;
  Index_New := 1;
  New_List.Length := List_1.Length + List_2.Length;
  loop -- Each iteration one element is moved into New_List
    -- Exit when one array is "empty"
    exit when Index_1 > List_1.Length or Index_2 > List_2.Length;
    if List_1.Items(Index_1) < List_2.Items(Index_2) then
      -- Copy from List_1
      New_List.Items(Index_New) := List_1.Items(Index_1);
      Index_1 := Index_1 + 1;
    else
      -- Copy from List_2
      New_List.Items(Index_New) := List_2.Items(Index_2);
      Index_2 := Index_2 + 1;
    end if;
    Index_New := Index_New + 1;
  end loop;
  -- Copy any remaining elements
  if Index_1 <= List_1.Items'Last then
    New_List.Items(Index_New .. New_List.Length) :=
      List_1.Items(Index_1 .. List_1.Length);
  else
    New_List.Items(Index_New .. New_List.Length) :=
      List_2.Items(Index_2 .. List_2.Length);
  end if;
  List_1.Length := 0;
  List_2.Length := 0;
end Merge_Lists;

```

(c) $O(N)$ for both implementations.

(d) The sequential version uses space for three complete lists. The linked version uses only the space needed for the actual elements. However, the linked version requires space for pointers as well as data.

26.



29. type List_Array is array (Positive range <>) of Element_Type

```

type List_Type (Max_Size : Positive) is
  record
    Count      : Natural := 0;  -- Length of List
    Current    : Natural := 0;  -- Designates "current" node
    Items      : List_Array (1..Max_Size);
  end record;

```

Chapter 9

1. Because its key is never examined. All list processing begins with the node that follows the header node.
3. (a) The special case of changing the external list pointer when a node is added at or deleted from the beginning of the list.
 - (b) The check for end of list is eliminated from the search procedure.
 - (c) No. Pushing or popping always changes the external stack pointer. There are no special cases to eliminate.
 - (d) Yes. The special cases of adding a new node to an empty queue and removing the last node from a queue can be eliminated with dummy nodes.

7. package Ptr_Stack is new Stack (Element_Type => Node_Ptr);

```

procedure Print_Reverse (List : in List_Type) is
  Stack : Ptr_Stack.Stack_Type;
  Ptr   : Node_Ptr;
begin
  if List /= null then
    Ptr := List.all.Next;
    loop -- Each iteration, an alias to a node is pushed.
      Ptr_Stack.Push (Stack, Ptr);
      exit when Ptr = List;
      Ptr := Ptr.all.Next;
    end loop;
    loop -- Each iterations, an alias is popped and printed.
      exit when Ptr_Stack.Empty (Stack);
      Ptr_Stack.Pop (Stack, Ptr);
      Print_Info (Ptr.all.Info);
    end loop;
  end if;
end Print_Reverse;

```

```

9. (c) procedure Traverse (List : in out List_Type) is
  -- This procedure does not allow modifications to list elements.
  Info      : Element_Type;
  Current_Is_Header : Boolean;
begin
  Reset (List);
  loop
    Retrieve_Current (Info, Current_Is_Header);
    exit when Current_Is_Header;
    Process (Info);
  end loop;
end Traverse;

```

11. (a) C.all.Next.all.Info
- (b) B.all.Back.all.Info
- (c) A.all.Next.all.Next
- (d) C.all.Back.all.Next

- (e) B.all.Back.all.Back.all
- (f) C.all.Back.all.Back
- (g) A.all.Back

13. (a) A Create procedure is required: the initial state (an empty list) cannot be achieved with default values for the record fields.

```
(b) procedure Create (List : in out List_Type) is
-- Precondition : List has not already been created
begin
    List := new Node_Type;
    List.all := (Info => Max_Element,
                Back => List,
                Next => List);
end Create;
```

15. (a) Doubly linked

(b) Circular

(c) Header and trailer

```
18. procedure Insert (List : in out List_Type;
                    Item : in Element_Type) is
    Found : Boolean;
    Pred_Loc : Node_Ptr;
    Location : Node_Ptr;
begin
    Search_Linked_List (List => List,
                        Key   => Key_Of (Item),
                        Found  => Found,
                        Pred_Loc => Pred_Loc,
                        Location => Location);
    loop -- loop to end of duplicates
        exit when Key_Of (Item) < Key_Of (Location.all.Info);
        Pred_Loc := Location;
        Location := Location.all.Next;
    end loop;
    Pred_Loc.all.Next := new Node_Type' (Info => Item,
                                         Next => Location);
end Insert;
```

20. To prevent the memory leaks and aliases that occur when the assignment operator is used.

Chapter 10

2. (a) John

Nell

Susan

Suzanne

(b) David

Naomi

Robert

(c) 8

1

9

4. Because several stacks can share the same array, we need only choose an upper bound on the number of *all* stack elements, not on each stack.

6. The `Direct_IO Size` function returns the number of components in the file. Because of deletions, not all components in the file are part of the list.

Chapter 11

1. (a) The base case is a nonrecursive exit from the recursive routine.
- (b) The general (or recursive) case is a path that includes a recursive call to the routine to solve a smaller version of the original problem.
- (c) The run-time stack is a structure that keeps track of the activation records at run time in order to preserve the values of parameters, return addresses, registers, and so on.
- (d) Binding time refers to the point in the compile/execution cycle when variable names are associated with addresses in memory.
- (e) Tail recursion occurs when the recursive call is the last statement executed in a recursive procedure or function.
- (f) Binding is when variable names are associated with addresses in memory.

4. Answering yes to Question 1 provides us with a base case that works correctly. In answering Question 3, we make an assumption that the function will work for some arbitrary case; we can then show that applying the function to the next value results in the correct answer in the general case.

5. A stack.

7. The base cases and general case are commented in the function.

```
function Puzzle (Base : in Natural;
                Limit : in Natural) return Integer is
    Result : Integer;
begin -- Puzzle
    if Base > Limit then
        Result := -1;                                -- Base Case 1
    elsif Base = Limit then
        Result := 1;                                  -- Base Case 2
    else
        Result := Base * Puzzle (Base + 1, Limit);    -- General Case
    end if;
    return Result;
end Puzzle;
```

8. (a) -1
- (b) 120
- (c) 1

12. An operand that is an integer.

13. (a) This answer is incorrect; it resets the value of Sum_Squares to 0 inside the recursive function. The assignment of Sum_Squares to 0 should occur in the base case (when List = null). Half credit is awarded for getting the general case correct, though.

(b) This answer correctly calculates the sum of the squares, but it gets no credit because the assignment was to write a *recursive* function. This is an iterative (looping) solution.

(c) Correct recursive solution. Full credit is awarded.

(d) This solution is functionally equivalent to (c); it just avoids the last recursive call (to an empty list) by setting Sum_Squares to the square of the last list element in the base case. This answer runs into problems if the list is empty, for List.Next is accessed in the *if* condition. However, the specification said that we can assume that the list is not empty, so full credit is awarded.

(e) This solution is incorrect; the general case will not correctly calculate the sum of the squares. Quarter credit is given, though, for using the correct control structure (*if*) for a recursive solution and for getting the base case correct.

14. (a) The recursive solution is inefficient because some of the intermediate values are calculated more than once.

(c) Iterative solution:

```
function Fibonacci (N : in Positive) return Positive is
  Fib_N_2 : Positive; -- Fibonacci (N-2);
  Fib_N_1 : Positive; -- Fibonacci (N-1);
  Result  : Positive;
begin
  if N <= 2 then
    Result := 1;
  else
    Fib_N_2 := 1;
    Fib_N_1 := 1;
    for Count in 3..N loop
      Result := Fib_N_2 + Fib_N_1;
      Fib_N_2 := Fib_N_1;
      Fib_N_1 := Result;
    end loop;
  end if;
  return Result;
end Fibonacci;
```

18. (a) It is difficult to establish that the recursive calls satisfy Question 2—that they are moving toward the base case.

(b) Ulam (7) makes 16 recursive calls.

Ulam (8) makes 3 recursive calls.

Ulam (15) makes 17 recursive calls.

19. Dynamic storage allocation provides a separate workspace for each invocation of a procedure or function, which is a requirement for recursion.

20. Binding time refers to the point in the compile/execute cycle when variable names are associated with addresses in memory. For recursion to be possible, variables must be bound to addresses at run time, not at compile time.

23. (a) False. Recursive solutions are often less efficient in terms of computing time.

(b) True—assuming that the recursive solution is not much less efficient.

(c) False. Recursive solutions generally require more space in the run-time stack.

(d) True. (Don't you want a good grade in this course?)

24. The base cases occur when the current position has already been tried, or is the exit or a trap, or when we are already free from the maze. The general case occurs in every other case.

Chapter 12

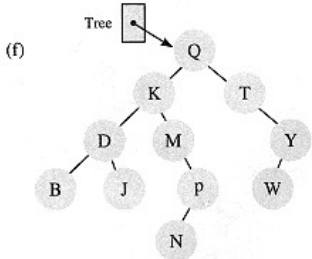
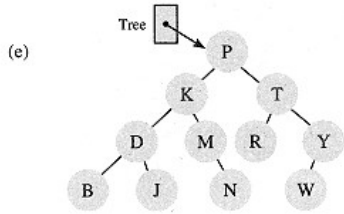
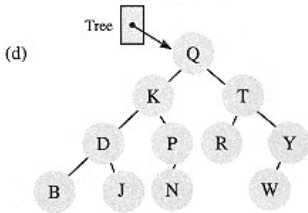
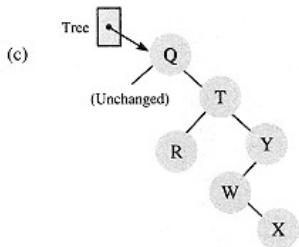
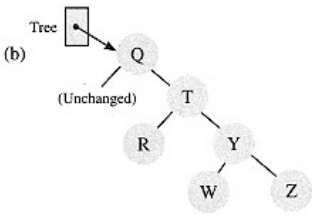
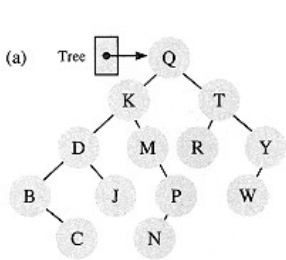
2. (c) $2^{N+1} - 1$.

4. A node in the N th level (where the root level = 0) has N ancestors.

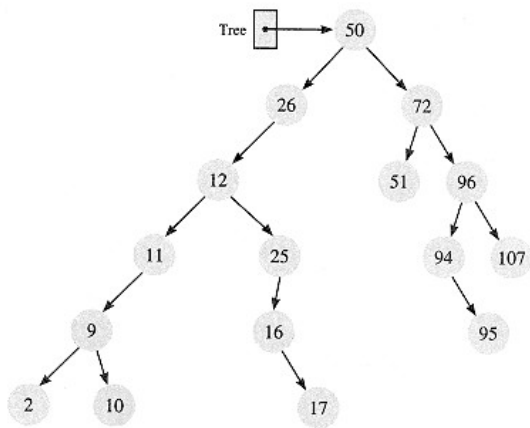
5. (a) 30 (5 different shapes with 6 different combinations of values for each)

(b) 5 (5 different shapes with one legal combination of values for each)

9.



11.



13. (a) The path to search for 61 would go through the nodes containing 56, 69, 59, 62, and 61.

(b) The path to search for 28 would go through the nodes containing 56, 47, 22, 29, and 23, before determining that 28 is not in the tree.

17. (a) False (c) True
(b) False (d) False

18. No. An inorder traversal would leave the elements in a sorted order; when you went to rebuild the tree, you would have a highly skewed tree—one with only one node per level.

23. (a)

```

procedure Find_And_Unlink_Min (Tree    : in out Node_Ptr;
                               Min_Ptr : out Node_Ptr) is
-- Finds and unlinks the node with the minimum key from the tree

-- Preconditions : Tree is not empty.

-- Postconditions : Min_Ptr designates the node containing the
--                  smallest Key in Tree.
--                  The node designated by Min_Ptr is unlinked
--                  from the tree.
begin
  if Tree.all.Left = null then    -- Is there a left child?
    -- Base case, root contains the minimum key in Tree
    Min_Ptr := Tree;              -- Return pointer to it
    Tree    := Tree.all.Right;    -- Unlink it from Tree
  else
    -- General case, keep looking in the left subtree
    Find_And_Unlink_Min (Tree => Tree.all.Left, Min_Ptr => Min_Ptr);
  end if;
end Find_And_Unlink_Min;

```

(b)

```

procedure Delete_Root (Tree : in out Tree_Type) is
-- This procedure deletes the root node from the given tree.

-- Preconditions: Tree is not empty.

-- Postconditions: The root node of Tree is deleted.
--                 Tree remains a binary search tree.

    To_Recycle : Node_Ptr; -- For recycling nodes
    Succ_Ptr   : Node_Ptr; -- Designates the root's logical successor
begin
    if Tree.all.Left = null and Tree.all.Right = null then
        -- Root node has no children
        Free (Tree); -- Entire tree now empty
    elsif Tree.all.Left = null then
        -- Root node has only a right child
        To_Recycle := Tree; -- Save for later deallocation
        Tree := Tree.all.Right; -- Unlink the root node
        Free (To_Recycle); -- Deallocate former root node
    elsif Tree.all.Right = null then
        -- Root node has only a left child
        To_Recycle := Tree; -- Save for later deallocation
        Tree := Tree.all.Left; -- Unlink the root node
        Free (To_Recycle); -- Deallocate former root node
    else -- Root node has two children
        -- Find and unlink the logical successor (The smallest key in
        --                                     the right subtree)
        Find_And_Unlink_Min (Tree => Tree.all.Right,
                             Max_Ptr => Succ_Ptr);
        Tree.all.Info := Succ_Ptr.all.Info; -- Copy Info from successor
        Free (Succ_Ptr); -- Deallocate successor
    end if;
end Delete_Root;

```

```

27. (a) procedure Ancestors (Tree : in Tree_Type;
                             Key   : in Key_Type) is
    begin
        if Key_Of (Tree.all.Info) /= Key then
            -- General Case: process this node and then
            -- process the rest of the ancestors.
            Process (Element => Tree.all.Info);
            if Key_Of (Tree.all.Info) < Key then
                Ancestors (Tree => Tree.all.Right, Key => Key);
            else
                Ancestors (Tree => Tree.all.Left, Key => Key);
            end if;
            -- else Base Case: Node containing Key is found
            --                 so stop processing
        end if;
    end Ancestors;

```



```

28. procedure Ancestors (Tree : in Tree_Type;
                        Key  : in Key_Type) is
begin
    if Key_Of (Tree.all.Info) = Key then
        -- General Case: process the rest of the ancestors
        -- and then process this node.
        if Key_Of (Tree.all.Info) < Key then
            Ancestors (Tree => Tree.all.Right, Key => Key);
        else
            Ancestors (Tree => Tree.all.Left, Key => Key);
        end if;
        -- Process this node
        Process (Element => Tree.all.Info);
    -- else Base Case: Node containing Key is found
    -- so stop processing.
    end if;
end Ancestors;

32. (a) procedure Copy_Tree (Tree      : in Tree_Type;
                           Duplicate : out Tree_Type);
    -- Creates a copy of Tree
    --
    -- Preconditions : None
    --
    -- Postconditions : Duplicate is a binary tree that is
    --                  identical to Tree.

    (b) procedure Copy_Tree (Tree      : in Tree_Type;
                           Duplicate : out Tree_Type) is
    begin
        if Tree = null then
            Duplicate := null; -- Base case
        else -- General case
            -- Create the root node of the duplicate tree
            Duplicate := new Node_Type;
            Duplicate.all.Info := Tree.all.Info;
            -- Create the copy of the left subtree
            Copy_Tree (Tree      => Tree.all.Left,
                      Duplicate => Duplicate.all.Left);
            -- Create the copy of the right subtree
            Copy_Tree (Tree      => Tree.all.Right ,
                      Duplicate => Duplicate.all.Right );
        end if;
    end Copy_Tree;

34. (a) procedure Mirror_Tree (Tree  : in Tree_Type;
                              Mirror : out Tree_Type);
    -- Creates a mirror-image copy of Tree
    --
    -- Preconditions : None
    --
    -- Postconditions : Mirror is a binary tree that is
    --                  a mirror-image of Tree.

```

```

(b) procedure Mirror_Tree (Tree   : in Tree_Type;
                          Mirror : out Tree_Type) is
begin
    if Tree = null then
        Mirror := null;  -- Base case
    else -- General case
        -- Create the root node of the mirror tree
        Mirror := new Node_Type;
        Mirror.all.Info := Tree.all.Info;
        -- Create the mirror-image of the left subtree
        Mirror_Tree (Tree => Tree.all.Left,
                    Mirror => Mirror.all.Right);
        -- Create the mirror-image of the right subtree
        Mirror_Tree (Tree => Tree.all.Right,
                    Mirror => Mirror.all.Left);
    end if;
end Mirror_Tree;

```

(c) This tree can be used for binary searching, if the search conditions are reversed:

```

if Key < Key_Of (Tree.all.Info) then
    Search the right subtree
else
    Search the left subtree
end if;

```

37.

```

procedure Raise_One (Employee : in out Employee_Rec) is
begin
    case Employee.Appraisal is
        when Superior =>
            Employee.Salary := 1.10 * Employee.Salary;
        when Good =>
            Employee.Salary := 1.05 * Employee.Salary;
        when Fair =>
            Employee.Salary := 1.03 * Employee.Salary;
        when Warning =>
            null;
        when Pink_Slip =>
            Employee.Salary := 0.0;
    end case;
end Raise_One;
procedure Raise_Time is new Emp_Tree.Traverse (Process => Raise_One);

```

40.

```

Package Student_Stack is new Stack (Element_Type => Student_Rec);

```

```

procedure Print_Senior_Women (Tree : in GPA_Tree.Tree_Type) is

    Stack   : Student_Stack.Stack_Type;
    Student : Student_Rec;

    procedure Save_If_Senior_Woman (Student : in out Student_Rec) is
    begin
        if Student.Class_Type = Senior and Student.Sex = Female then
            Student_Stack.Push (Stack, Student);
        end if;
    end Save_If_Senior_Woman;

```

```

    end if;
end Save_If_Senior_Woman;

procedure Check_All_Students is new GPA_Tree.Traverse
    (Process => Save_If_Senior_Woman);
begin -- Print_Senior_Women
    -- Push all women seniors in order by GPA onto the stack.
    Check_All_Students (Tree => Tree, Order => GPA_Tree.Inorder);
    loop
        exit when Student_Stack.Empty (Stack);
        Student_Stack.Pop (Stack, Student);
        Text_IO.Put (Student.Name);
        Float_IO.Put (Item => Student.GPA, Fore => 3, Aft => 2, Exp => 0);
        Integer_IO.Put (Student.ID_Num);
    end loop;
end Print_Senior_Women;

```

43.

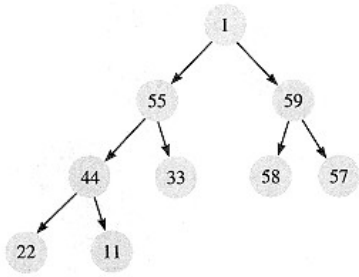
Nodes	.Info	.Left	.Right	Tree	Free
(1)	Q	2	3	1	9
(2)	L	4	5		
(3)	W	6	0		
(4)	F	0	0		
(5)	M	0	7		
(6)	R	0	8		
(7)	N	0	0		
(8)	S	0	0		
(9)	?	10	?		
(10)	?	11	?		

44.

Nodes	.Info	.Left	.Right	Tree	Free
(1)	Q	2	3	1	6
(2)	L	4	5		
(3)	W	8	0		
(4)	F	9	0		
(5)	M	0	7		
(6)	R	10	8		
(7)	N	0	0		
(8)	S	0	0		
(9)	B	0	0		
(10)	?	11	?		

Chapter 13

1. (a) Trees (b), (d), and (e) are complete.
 (b) Trees (b) and (e) are full.
 (c) Trees (b) and (d) are heaps.
4. (a) Note that NumElements = 9; thus 99 is not in the tree.

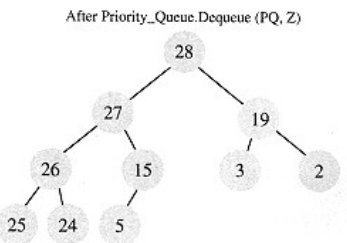
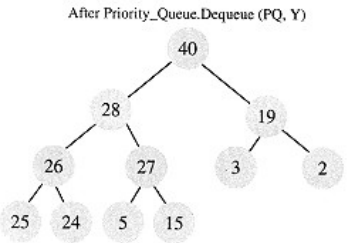
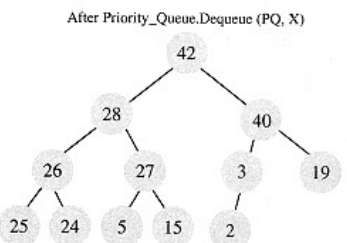
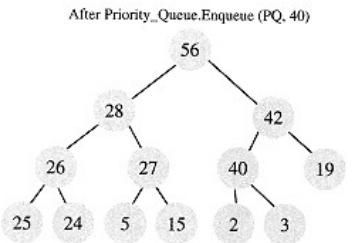
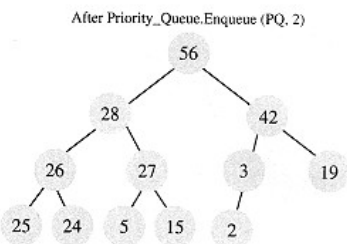
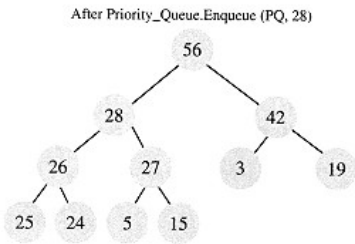


- (b) The tree is not a heap. The following procedure call makes the tree into a heap.

```
Reheap_Down (Heap => Heap (1..9));
```

7. By supplying an actual function "<" for the formal generic function ">".

11. (a)



(b) X = 56, Y = 42, Z = 40.

```

14. (a) private
    -- An unconstrained array of priority queue elements
    type Queue_Array is array (Positive range <>) of Element_Type;

    type Queue_Type (Max_Size : Positive) is
        record
            Count : Natural := 0;
            Items : Queue_Array (1..Max_Size);
        end record;
    end Priority_Queue;
(c) procedure Dequeue (Queue : in out Queue_Type;
                      Item : out Element_Type) is
begin
    if Queue.Count = 0 then
        raise UNDERFLOW;
    else
        Item := Queue.Items(1);
        Queue.Items(1..Queue.Count-1) := Queue.Items(2..Queue.Count);
        Queue.Count := Queue.Count - 1;
    end if;
end Dequeue;

```

15. (a) The highest priority element is the one with the largest time stamp. (This assumes that time stamps never reach Natural 'Last.)

(b)

```

type Stack_Rec is
    record
        Element : Element_Type;
        Time_Stamp : Natural;
    end record;

function Stamp_Of (Element : Stack_Rec) return Natural is
begin
    return Element.Time_Stamp;
end Stamp_Of;

package Stack is new Priority_Queue (Element_Type => Stack_Rec,
                                     Priority_Type => Natural,
                                     Priority_Of => Stamp_Of,
                                     ">" => Standard.">");

```

(c) Push
Assign the new element the next largest time stamp.
Enqueue the new element into the priority queue.

Pop
Dequeue an element from the priority queue.

(d) Assuming a heap is used to implement the priority queue. The Push operation is $O(\log_2 N)$; the new element is added as a leaf and then moved up through the tree to become the new root (Reheap_Up). The Pop operation is also $O(\log_2 N)$; the root element is replaced by the last element which is then moved down through the tree (Reheap_Down). These operations do not compare favorably with the $O(1)$ operations developed in Chapter 6.

17. (a) The string length is used to determine their priority (longer length = higher priority).

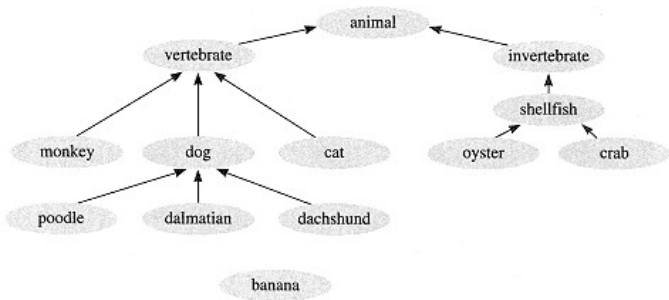
(b)

NumElements = 10

(1)	"introspective"
(2)	"interviewing"
(3)	"intellectual"
(4)	"intimate"
(5)	"intelligent"
(6)	"interesting"
(7)	"internal"
(8)	"into"
(9)	"in"
(10)	"intro"
(11)	"intensive"

23. "Works with" is the best description of the relationship represented by the edges between the vertices in Employee_Graph, because it is an undirected graph. The other relationships listed have an order implicit in them.

24.



26. (a) yes

(b) yes

(c) no

(d) no

(e) yes

(f) no

27. "is a kind of." For instance, dalmatian is a kind of dog.

29. (a) no

(b) yes

(c) Texas

```

33. (a) procedure Delete_Vertex (Graph : in out Graph_Type;
                                Key   : in   Key_Type);
-- Deletes the vertex with the given Key from the Graph
-- Preconditions : None

-- Postconditions : The vertex with the given Key is deleted
--                  from the graph. Any edges connected to this
--                  vertex are also deleted.

-- Exceptions      : VERTEX_ERROR is raised, if no vertex with the
--                  given key exists.
--                  The graph is not changed.

```

(b) Deleting a vertex is more complicated than deleting an edge because, in addition to removing the vertex from $V(\text{Graph})$, we must also remove the edges to all its adjacent vertices from $E(\text{Graph})$.

35. (a) Base Cases: (1) $\text{Start\_Vertex} = \text{End\_Vertex}$, (2) Start_Vertex has already been tried, (3) no more untried vertices adjacent to Start_Vertex .

General Case: Search all the untried adjacent vertices.

(b) Depth First Search

```

if Start_Vertex = End_Vertex then
    Found := True (Base Case #1)
elsif Start_Vertex is marked then
    Do nothing (Base Case #2)
else
    Mark Start_Vertex
    Get list of vertices adjacent to Start_Vertex
    if list contains one or more untried adjacent vertices
        Depth First Search all untried adjacent vertices
    else
        Do nothing (Base Case #3)
    end if
end if
end if

```

(c) This recursive procedure is unusual in that it contains a loop. The loop is used to process all of the nodes in a list. See if you can replace this iterative list traversal with a recursive traversal.

```

procedure Depth_First_Search (Graph      : in out Air_Routes.Graph_Type;
                             Start_City : in   City_String;
                             End_City   : in   City_String) is

    Found : Boolean; -- True when End_City is found

-- Nested recursive procedure
procedure Search (Graph      : in out Air_Routes.Graph_Type;
                 Start_City : in   City_String;
                 End_City   : in   City_String;
                 Found      : in out Boolean) is

-- List stores vertices adjacent to the starting city
List : Air_Routes.Key_List (Max_Vertices => 10);

```



```

begin -- Search
  if Start_City = End_City then
    Text_IO.Put (Start_City);
    Found := True;
  elsif not Air_Routes.Marked (Graph, Start_City) then
    -- Mark city as visited
    Air_Routes.Mark_Vertex (Graph => Graph, Key => Start_City);
    -- Display this city
    Text_IO.Put_Line (Start_City);
    -- Get the vertices adjacent to Start_City
    Air_Routes.Get_Adjacent_Vertices (Graph => Graph,
                                     Key => Start_City,
                                     Adj_Keys => List);

    -- Search all untried adjacent cities
    for Index in 1..List.Length loop
      if not Found and not Air_Routes.Marked
        (Graph => Graph,
         Key => List.Items(Index)) then
        Search (Graph => Graph,
               Start_City => List.Items(Index),
               End_City => End_City,
               Found => Found);
      end if;
    end loop;
  end if;
end Search;

begin -- Depth_First_Search
  Air_Routes.Clear_All_Marks (Graph);
  Found := False;
  Search (Graph, Start_City, End_City, Found);
end Depth_First_Search;

```

36. In the adjacency matrix below, if $\text{Alpha\_Graph}(V1, V2) = \text{True (T)}$, there is an edge between vertex $V1$ and vertex $V2$; if $\text{Alpha\_Graph}(V1, V2)$ is False (F) , there is no edge between them.

	(A)	(B)	(C)	(D)	(E)	(F)	(G)	(H)	(I)	(J)
(A)	F	T	T	T	F	T	F	F	F	F
(B)	T	F	T	F	T	F	T	F	F	F
(C)	T	T	F	F	F	T	T	F	T	F
(D)	T	F	F	F	F	T	F	T	F	F
(E)	F	T	F	F	F	F	T	F	F	T
(F)	T	F	T	T	F	F	F	T	T	F
(G)	F	T	T	F	T	F	F	F	T	T
(H)	F	F	F	T	F	T	F	F	T	F
(I)	F	F	T	F	F	T	T	T	F	T
(J)	F	F	F	F	T	F	T	F	T	F

Chapter 14

2. (a)

66	43	19	23	18	4	10	5	7	14
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)

(b)

18	14	10	5	7	4	19	23	43	66
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)

4. (a) bubble sort

(b) selection sort

6. (a) $O(N^2)$

(b) $O(N)$

(c) $O(N^2)$

(d) $O(N \log_2 N)$

(e) $O(N \log_2 N)$

9. Quick_Sort (c) would take the longest; Short_Bubble_Sort (b) would be the fastest.

11. Heap_Sort would be a better choice than Merge_Sort, because Merge_Sort needs a copy of the array for its processing whereas Heap_Sort sorts within the original array.

14. Only (b) is true.

16. Both (b) and (d) are true.

18. (a) For: Making Swap a procedure will make the code easier to read and understand.

Against: It would be more efficient, in terms of execution time, to code the swap algorithm "in line" instead of requiring a procedure call, which takes extra time to execute.

(b) Pragma INLINE allows us to have the efficiency of "in line" coded subprograms while still maintaining the ease of reading provided by procedural abstraction.

22. (a) procedure Selection_Sort (Info : in out Array_Type) is

```

    Current  : Positive;
    Smallest : Positive;
    Temp     : Element_Type;

begin -- Selection_Sort
    Current := Info'First;
    loop
        exit when Current >= Info'Last;
        -- Find the index of the smallest unsorted element
        Smallest := Current;
        for Index in Current + 1 .. Info'Last loop
            if Info(Index) < Info(Smallest) then
                Smallest := Index;
            end if;
        end loop;
    end loop;
end;
```

```

        -- Put the smallest element into the correct spot
        Temp      := Info(Current);
        Info(Current) := Info(Smallest);
        Info(Smallest) := Temp;
        -- Shrink the unsorted part of the array
        Current := Current + 1;
    end loop;
end Selection_Sort;

```

(b) Advantage: Because there are no procedure or function calls, the sort should execute more quickly.

Disadvantage: The code is not as readable.

```

25. procedure Insertion_Sort (Info : in out Array_Type) is
    -- Body 14-4
    .
    .
    .
end Insertion_Sort;

procedure Quick_Sort (Info : in out Array_Type
    Limit : in Positive) is

    Split_Index : Positive;

    -- All local subprograms identical to those in Body 14-6

    procedure Swap . . .

    function ">=" . . .

    procedure Split . . .

begin -- Quick_Sort
    if Info'Length > Limit then
        Split (Info, Split_Index);
        Quick_Sort (Info => Info(Info'First .. Split_Index - 1));
        Quick_Sort (Info => Info(Split_Index - 1 .. Info'Last));
    else
        Insertion_Sort (Info);
    end if;
end Quick_Sort;
28. function "<" (Left : in Card_Type; Right : in Card_Type)
    return Boolean is
begin
    if Left.Number = Right.Number then
        return Left.Suit < Right.Suit;
    else
        return Left.Number < Right.Number;
    end if;
end "<";

```

```

29. procedure Card_Sort is new Heap_Sort (Element_Type => Card_Type,
                                         Array_Type  => Card_Array,
                                         "<"         => "<");

```

33. Wilbur could use a modification of the selection sort: Each iteration would put the next largest element in place (rather than the next smallest, as shown in this chapter). He wouldn't have to sort the whole array, because he needs only the top 10. Therefore the outer loop would run from 1 to 10 rather than from 1 to Num_Pigs.

38. Change the `for` loop in procedure `Collect_Queue`s to reverse the order in which the queues are collected. The `for` loop becomes

```

for Queue_Index in reverse Queues'Range loop

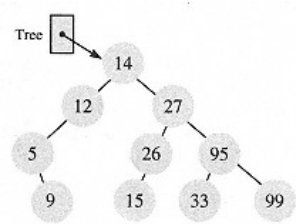
```

Chapter 15

1.

14	27	95	12	26	5	33	15	9	99
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)

3.



5.

HashTable	
(0)	140
(1)	620
(2)	
(3)	
(4)	
(5)	145
(6)	66
(7)	47
(8)	87
(9)	126
(10)	90
(11)	285
(12)	467
(13)	153
(14)	393
(15)	395
(16)	566
(17)	177
(18)	735
(19)	

7.

HashTable			
(0)	90	140	620
(1)			
(2)			
(3)	153	393	
(4)			
(5)	145	285	395
(6)	66	126	566
(7)	47	87	177
(8)	467	735	
(9)			

9.

Number of Comparisons				
Value	Problem 5	Problem 6	Problem 7	Problem 8
66	1	1	1	1
467	6	6	4	4
566	11	3	3	3
735	4	2	11	4
285	7	2	2	2
87	2	2	2	2

11. (a) 100

(b) 50

(c) 50

13. (a) 50

(b) 50

(c) 50

15. (a) False; a binary search of an ordered set of elements in an array is usually faster than a sequential search of the elements, but it will not be faster if the number of elements is very small.

(b) False; a binary search is an $O(\log_2 N)$ algorithm.

(c) False; a binary search of elements in an array requires only that the elements be sorted; they may be ordered from smallest to largest or vice versa. (Of course, the algorithm will be determined by the ordering scheme.)

(d) True.

(e) False; this statement would be true only if we could guarantee that there would be no collisions. If there are collisions, the order in which elements are added determines their place in the array.

(f) False; increasing the size of the array will reduce the number of collisions only if the hash function will distribute the elements throughout the larger array.

(g) False; we will still need to resolve collisions when the buckets are full.

(h) True.

(i) False; the procedures in this chapter are used only for internal searching.

(j) True.

19. Arrange the most commonly used reserved words toward the beginning of the list. For instance, reserved words like `if`, `then`, `else`, `is`, and `end` are commonly used. On the other hand, `abort`, `xor`, `terminate`, and `goto` are rarely used at all. These less common reserved words would be placed at the end of the list.

21. (a) This change would not be any better than the original hash function but would probably work well. It requires extra time to determine whether or not each letter is a consonant.

(b) This change would be okay.

(c) This would not be a good modification, because the likelihood of collisions is increased.

(d) This would not be a good modification, because the likelihood of collisions is increased.

(e) This change would be okay, because the variety of characters would probably give a good hash address.

(f) This change would not be good, because the hash address would be calculated from the blank padding for strings that are shorter than the maximum length.

(g) This change would not be good, because the product of the integer version of all the letters in the string would likely be greater than `Integer.Last`.

24. Key mod 17

27. (d) only

Index

A

Abbreviations for identifiers, 771

`abs` operator. precedence of, 803

Abstract data objects (ADOs)

- for Bingo card, 102-108
- initializing, 321

Abstract data types (ADTs), 15, 108-116

Abstract level

- of binary search trees, 534-543
- of Bingo card, 100-101
- of classes, 80
- of data structures, 96-97
- of FIFO queues, 268-271
- of file-based sorted lists, 454-455
- of graphs, 615-622
- of linear lists, 308-314
- of one-dimensional arrays, 88-90
- of priority queues, 608-610
- or records, 82-83
- of sets, 130-135
- of stacks, 222-226
- of strings, 176-180
- of two-dimensional arrays, 93-95
- of unbounded-length strings, 202-205

Abstraction in program design, 10-12, 14

Access type variables, 193-194

- accessing data with, 195-196
- exceptions with, 198
- with FIFO queues, 284
- as local variables, 205
- new allocator for, 194-195
- null values for, 194, 198, 377
- saving, 438
- with structured types, 198-200
- `Unchecked_Deallocation` with, 196-198

Access types

- initializing, 377
- for strings, 193-200

Accessing

- access type variables for, 195-196
- array elements, 88-89, 92-95
- records, 82-83, 85, 87-88, 800

`Activation_Rec` record, 491

Activation records, 490-492

- for parameters, 496
- run-time stacks for, 492-496

Active error checking, 36-37

Actual parameters, 143, 145, 785, 806

Ada 83, strings in, 191-192

Ada 95

- child units in, 790
- strings in, 215-216

Ada Language Reference Manual, 795

`Ada.Numerics.Discrete_Random` package, 792

`Ada.Strings.Bounded` package, 176

`Ada.Strings.Fixed` package, 176

`Ada.Strings.Unbounded` package, 176, 216

`Add_Edge` procedure (`Graph`), 620, 636

`Add` procedure (`Address_Book`), 318, 324

`Add` procedure (`Client`), 547, 552

`Add` procedure (`Unbounded_Natural`), 412, 415, 419-420

`Add_Vertex` procedure (`Graph`), 620, 634, 636

Adding set elements, 131-132

Addition

- in postfix expressions, 227
- precedence of, 803

'`ADDRESS`' attribute, 776

`Address_Book` package, 317-326

`Address_Rec` record, 318, 546

Addresses. *See also* Memory and memory management

- for arrays, 90-92, 439
- attribute for, 776
- in dynamic storage allocation, 490
- for records, 85
- stacks for, 234
- in static storage allocation, 488-490

Adequate function, 170

Adjacency lists, 638-640

Adjacency matrices, 632-634

Adjacent vertices, 616

Adjectives for identifiers, 771

Advanced iterators, 406-409

'`AFT`' attribute, 776

Aggregates

- array, 803
- record, 800-801

`Air_Routes` package, 624

Airline routes, graphs for, 622-632

Algorithms

- comparing, 20-26
- in software engineering, 3

Aliases, 196-198, 201

`.all` in dot notation, 195-196, 199-200

Allocators, 194-195

Alphabetizing names, 180-184

Ampersand (`&`), precedence of, 803

Ampersand function (`&`) (`Bounded_Strings`), 177-178, 187-188

Analyses

- of algorithms, 21-26
- verification activities in, 58

Ancestors in binary trees, 535

`and` operator, 139, 803

`and then` operator, 331

Apostrophe (`'`) for attributes, 799

`Append` procedure (`Unbounded_Natural`), 417

Application level

- for binary search trees, 543-554
- for bounded-length strings, 180-185
- for classes, 80
- for data structures, 96
- for FIFO queues, 271-277
- for graphs, 622-632
- for linear lists, 314-326
- for one-dimensional arrays, 90
- for priority queues, 610-611
- for records, 83-84
- for sets, 135-137
- for stacks, 226-234
- for two-dimensional arrays, 95
- for unbounded-length strings, 205-208

Archer, Gwilym, 792

Area codes in records, 317

Arrangement in binary trees, 537

Array descriptors, 90, 92

Arrays, 801

- abstract level of, 88-90, 93-95

- access variables with, 198-200

- advantages of, 438

- application level of, 90, 95

- of arrays, 93-94

- attributes of, 802

- constrained, 801

- declaration of, 796

- versus files, 453

- formal and actual types for, 151-152

- for graphs, 632-638

- implementation level of, 90-93, 95

- indexes for, 88-89, 92-93, 158-163

- for linear lists, 327-342, 358-361

- for lists, 83-84

- memory management for, 90-92, 327, 439-440, 448-450

- in merge sorts, 668-671

- for nonlinked binary search trees, 598

- one-dimensional, 88-93

- operations on, 802

- as parameters, 808

- of pointers, sorting, 689-691

- for queues, 278-283, 290-294

- in quick sorts, 674-678

- in radix sorts, 692-697

- of records, linked lists as, 438-452

- for sets, 137-139

- sorting, 155-156

- for stacks, 235-240

- two-dimensional, 93-95

- unconstrained, 801-802

Arrow symbol (`=>`) for parameters, 806

ASCII character set

- list of, 752-753

- in `Standard`, 765-766

Asking questions about specifications, 34

Assertions

- in design, 37-38

- with loop invariants, 40

`Assign` procedure (`Unbounded_Natural`), 412, 419

`Assign` procedure (`Unbounded_Strings`), 203, 210, 212-214

Assignment statements, 804

Assignments with access variables, 201

Associations

- with arrays, 803

- with parameters, 806

- with records, 800

Assumptions

- checking, 43

- in design, 38-39

- run-time errors from, 36

- in specifications, 9

Asterisk (`*`), precedence of, 803

Asterisk (`*`) function (`Bingo_Number_Set`), 133, 140

Asterisk (`*`) function (`Discrete_Set`), 144, 148

Asterisk (`*`) function (`Float_Array_Ops`), 153

Asterisk (`*`) function (`Fraction_Class`), 53

Atomic values, 80

Attributes, 799-800

- for arrays, 802

- for classes, 15

- predefined, 776-783

`Auto_Part_List` package, 407-409

B

Backtracking techniques, 504

`Backup` procedure (`Client`), 548, 553

Backus, John, 740

Backus-Naur Form (BNF), 740

Balanced binary trees, 562-563

Base addresses

- for arrays, 90-92

- for records, 85

'`BASE`' attribute, 776

Base-case question in recursion, 476-477

Base type, attribute for, 776

Benchmark tests, 22-23

Big-O notation, 22-26, 59-60

Binary files, 320, 452, 810

Binary search case study, 59-70

`Binary_Search` procedure

- final version, 69-70

- as generic package, 157-166

- recursion for, 485-487

- test drivers for, 63-66

- version 1, 62-63

- version 2, 68-69

`Binary_Search_Tree` package, 538-541

Binary search trees

- abstract level of, 534-543

- application level for, 543-554

- clearing, 569

- deleting elements in, 562-568, 575-577

- heaps, 598-608

- implementation level of, 554-578

- inserting elements into, 557-562, 572-574

- iterative operations on, 570-578

- modifying elements in, 556-557, 557

- nonlinked, 596-580

- performance of, 578-580

- recursive operations on, 555-569

- retrieving elements in, 556-557, 577

- searching, 570-572

- size operations on, 555-556

- specification for, 538-541

- traversing, 541-543, 568-569

Binary searches, 712-713

- algorithms for, 61-62

- Big-O notation for, 60

- efficiency of, 23

Binding memory with variables, 487

Binding time, 487

`Bingo_Basket` package, 118-119

Bingo card, 99-116

`Bingo_Card_ADO` package, 102-108

`Bingo_Card_ADT` package, 109-116

`Bingo_Definitions` package, 101-102

Bingo game case study, 117-123

`Bingo_Number_Set` package, 132-135, 140-142

`Bingo_Simulation` procedure, 120-121

`Birthdays` procedure (`Address_Book`), 319-320, 325

Bits allocated, attribute for, 782

Black box testing, 50

Blank lines with inline comments, 769-770

Blanks from strings, removing, 184-185

BNF (Backus-Naur Form), 740

Bodies for packages, 17-18

Boolean operators and expressions

- for arrays, 90

- for loops, 40

- with recursion, 477-489

- short-circuit evaluation of, 331

Boolean type, 799

- as atomic class, 80

- for sets, 137-139

Bottom-up testing, 53-55

Boundary conditions

- operations at, 38

- testing, 50

Boundary problem in recursion, 511

Bounded-length strings, 176

- application level for, 180-185

- records for, 185-186

- sorting, 180-184

- specification for, 177-180, 186-192

`Bounded_Strings` package, 177-180, 186-192

Bounds for arrays, 90-91

Box symbol (`<>`)

- for arrays, 86, 801

- for discrete type formal parameters, 146-147

Braces (`{ }`)

- in EBNF definitions, 740

- for sets, 130

- for vertices, 616

Brackets ([]) in EBNF definitions, 740

Branches

- with recursion, 475

- testing, 50-51

Breadth_First_Search procedure, 628

Breadth-first searching in graphs, 626-629

Break points, 56-57, 687

Bubble_Sort procedure, 662-663

Bubble sorts, 660-664

Bubble_Up procedure, 662-663

Buckets in hashing, 722-723

Budget considerations, 5-6

Bugs, source of, 32-37

Bushy binary trees, 562-563

C

Calculators using postfix expressions, 226-227

Calendar package

- for dates, 317

- specification for, 762

'CALLABLE' attribute, 776

Capitalization for identifiers, 771

Car_Rec record, 273

Card_Type type, 111

Caret (^) in ceiling function, 535

Case-sensitivity, 35-36, 798

Case statements, 804

Catenate procedure (Unbounded_Strings), 203, 209-210

Catenating strings, 176, 802

Catenation operators

- for arrays, 90

- in Bounded_Strings, 177-178, 187-188, 203

- in Unbounded_Strings, 203, 209-210

Ceiling function, 535

Cells, memory, 84-85

Chains in hashing, 722-724

Change_Name procedure (Address_Book), 319, 325

Change procedure (Address_Book), 319, 324

Change procedure (Client), 548, 553

Character input-output routines in Text_IO, 756

Character sets

- ASCII, 752-753

- in Standard, 765-766

Character type, 80, 799

Check_Cards procedure (Bingo_Simulation), 121

Check In services in medical emergency response application, 543

Check_Inventory procedure (Auto_Part_List), 407

Check_Inventory procedure (Part_List), 313-314

Check_One_Birthday procedure (Address_Book), 323

Check_One_Column procedure (Bingo_Card_ADO), 106-107

Check_One_Column procedure (Bingo_Card_ADT), 114

Check_One_Part procedure (Auto_Part_List), 407-408

Check_One_Part procedure (Part_List), 313

Check_One_Row procedure (Bingo_Card_ADO), 105-106

Check_One_Row procedure (Bingo_Card_ADT), 113

Check procedure (Client), 547, 551-552

Check_Stock procedure (Auto_Part_List), 408

Checklist for deskchecking, 43

Child units, 790

Children in binary trees, 534-536, 596-597

Circular arrays for FIFO queues, 280-281

Circular doubly linked lists

- finding elements in, 398-400

- operations on, 400-402

- for unbounded natural numbers, 413-414

Circular linked lists, 386-388

- deleting elements from, 395-397

- finding elements in, 389-392

- inserting elements into, 392-395

- for queues, 290

- traversing, 388-389

City_Stack package, 624

Clarity in recursion, 500-501

Classes, 15

- encapsulating, 253-254

- packages for, 18-20

- perspectives on, 80

Clauses, EBNF definition of, 784

Clear_All_Marks procedure (Graph), 621, 624, 635, 637

Clear box testing, 50

Clear operations

- for binary search trees, 569

- for FIFO queues, 270, 279, 286

- for linked linear lists, 344

- for sequential lists, 329

- for stacks, 223

Clear procedure (Binary_Search_Tree), 539

Clear procedure (Bingo_Card_ADO), 103-104

Clear procedure (Bingo_Card_ADT), 110, 112

Clear procedure (Graph), 620, 634-635

Clear procedure (Priority_Queue), 609, 614

Clear procedure (Queue), 270

- array-based, 283

- linked list-based, 288

Clear procedure (Sorted_List), 309-310

- for array-based linked list implementation, 446

- for file-based sorted list implementation, 455, 460

- for linked linear list implementation, 354-355

- for sequential list implementation, 340

Clear procedure (Stack), 224, 236-237, 247-248, 250

Client_IO package, 549-550

Client_Rec record, 546

Clients package

- implementation of, 548-554

- specification for, 545-548

Close procedure (Direct_IO), 452

Close procedure (Sorted_List), 454-455, 457, 463

Club_List record, 688

Clustering in hashing, 720

Code coverage in testing, 50-52

Code reuse, 15, 59, 116

- array attributes for, 802

- in Binary_Search procedure, 157-166

- determining, 320

- generic units for, 142-146

Coding

- memory for, 215

- verification activities in, 58

Coding phase, changes in, 5

Cohesive modules, 13

Collect_Queues procedure (Radix_Sort), 696

Collection classes, 95

Collections

- finite, 88

- unordered, 130

Collisions in hashing, 716-724

Colons (:) in EBNF definitions, 740

Column_Winner function (Bingo_Card_ADO), 107

Column_Winner function (Bingo_Card_ADT), 114-115

Columns

- in tables, 93

- Text_IO routines for, 755

Combinations function, 481-483

Commands, testing, 50

Comments, 768-770, 798

- for declarations, 772

- for identifier names, 43

- for loops, 40

- for postconditions, 101

Communication problems, 34

Comparing

- algorithms, 20-26

- records, 800

- sorts, 697-698

- strings, 189-190

Comparison operators for arrays, 802

Compilation units, 784-785

Compilations, 784, 787-790

Compile time

- errors at, 35-36

- variable binding at, 487

Compilers, 3

Complete binary trees, 597. *See also* Heaps

Complete graphs, 618

Complete programs, 4

Complex number type, 807-808

Complexity

- abstraction for handling, 10-12

- versus efficiency, 26

Composite types, 82

- one-dimensional arrays, 88-93

- records, 82-88

- two-dimensional arrays, 93-95

Computer programming, 2. *See also* Design: Programming style

Conditions

- boundary, 38, 50

- loop, 40-41, 332

Constant time, 23

Constants

- for array indexes, 89

- for ASCII characters, 752-753

- in Big-O notation, 24

- declarations for, 772, 796

- deferred, 135, 138

- identifiers for, 771

- in package specifications, 18

Constrained arrays, 796, 801

'CONSTRAINED' attribute, 776-777

CONSTRAINT_ERROR

- with access type variables, 198

- with array elements, 92

- exception handlers for, 36, 812

- in sequential searches, 331

- with string comparisons, 190

- in type checking, 799

Constraints, attributes for, 776-777

Constructors, 308-310

Context clauses, EBNF definition of, 784

Control characters in `Standard`, 766

Control routines in `Text_IO`, 755

Converting

- bounded-length and fixed-length strings, 179-180, 189, 204, 212

- numbers to bits, 728-729

- strings to numbers, 729

- types, 456, 803-804

Cookies, specification for, 9-10

Correct programs, 4

Correctness

- design for, 37-42

- functional, 20

Costs, debugging, 33-34

'COUNT' attribute, 777

Cover_Number procedure (`Bingo_Card_ADO`), 103, 105

Cover_Number procedure (`Bingo_Card_ADT`), 110, 113

Create procedure for doubly linked lists, 403

Critical loops, 21

Cubic time, 24

Curly braces (`{}`)

- in EBNF definitions, 740

- for sets, 130

- for vertices, 616

D

Dash (`-`)

- for comments, 798

- precedence of, 803

Data coverage in testing, 49-50

DATA_ERROR exception handlers, 36, 812

Data structures, 95-96

- abstract level of, 96-97

- access variables with, 198-200

- application level of, 96

- implementation level of, 97-99

- as parameters, 496

Data types. *See* Types

Dates, `Calendar` package for, 317

Debugging, 32.

- See also* Testing

- costs of, 33-34

- plans for, 56-57

- pragmas in, 687

- recursive routines, 497

Decimal digits, attributes for, 776-777

Declarations, 798

- for access variables, 194

- for arrays, 88, 90, 796

- for bounded-length strings, 180

- comments for, 769

- for constants, 796

- incomplete, 249

- for loop control variables, 805

- order of, 771-773

- for types, 795

- for variables, 796-797

Decomposition, function, 13-14

Default initial values, 796, 801

Deferred constants, 135, 138

Defining problems, 6-7

Definitions

- comments for, 769

- EBNF. *See* EBNF definitions

- in specifications, 8

Degenerate binary search trees, 579

Delegate languages case study, 167-172

Delete operations

- for binary search trees, 562-568, 575-577

- for circular linked lists, 395-397

- for doubly linked lists, 402

- with hashing, 719-720

- for heaps, 600-601

- for linked linear lists, 352

- for linked lists with dummy nodes, 385-386

- for sequential lists, 336-338

- for substrings, 184

Delete procedure

- for binary search trees, 564, 575

- for circular linked lists, 396-397

- for doubly linked lists, 402

- for linked lists with dummy nodes, 386

- for substrings, 184

Delete procedure (`Address_Book`), 318-319, 324

Delete procedure (`Binary_Search_Tree`), 539

Delete procedure (`Client`), 547, 553

Delete procedure (`Sorted_List`), 310

- for array-based linked list implementation, 446-447

- for file-based sorted list implementation, 455, 461

- for linked linear list implementation, 355-356

- for sequential list implementation, 340-341

Delete_Root procedure, 566, 576

Delivery, verification activities in, 58

'DELTA' attribute, 777

Delta value, attribute for, 777

Dependencies in compilation units, 784, 789

Dependency graphs, 789

Depth_First_Search procedure, 625-626

Depth-first searching in graphs, 622-626

Dequeue operations

- for FIFO queues, 269-270, 279-280, 285-286

- in priority queues, 608, 612

Dequeue procedure (Priority_Queue), 609, 613

Dequeue procedure (Queue), 270-271

- array-based, 283

- linked list-based, 289

Dereferencing

- of objects by access variables, 193, 195-196

- record fields, 243

Descendants in binary trees, 535

Descending order in sorts, 687-688

Descending_Sort procedure, 688

Design

- abstraction in, 10-12

- of Address Book, 316-317

- assertions in, 37-38

- for correctness, 37-42

- divide-and-conquer, 13-14

- errors in, 33-35

- of Escape program, 506-512

- function decomposition in, 13-14

- information hiding in, 12-13

- loop invariants in, 39-42

- object-oriented, 14-16

- preconditions and postconditions in, 38-39

- of unbounded natural numbers, 413-414

- verification activities in, 58

Design phase, changes in, 5

Designation of objects by access variables, 193

Deskchecking, 42-43

Diagonal_Winner function (Bingo_Card_ADO), 107-108

Diagonal_Winner function (Bingo_Card_ADT), 115

Difference of sets, 131-132, 139

Digit_Of function (Radix_Sort), 695

'DIGITS' attribute, 777

Digraphs, 616-618

Direct access to arrays, 88

Direct files, 452, 810

Direct_IO package, 452-453, 760-761, 810

Directed graphs, 616-618

Director objects, 119

Discrete_Random package, 792-794

Discrete_Set package, 143-149

Discrete types

- formal generic type parameter for, 146-149

- random number generator for, 792-794

Discriminants, record, 85-88

Display_Maze procedure (Escape), 515-516

Display_Results procedure (Escape), 516

Divide-and-conquer design, 13-14

Divide-and-conquer sorts, 668

Divide procedure (Unbounded_Natural), 412, 415-416, 422-423

Divider procedure, 46-48

Division

- in postfix expressions, 227

- by zero, 36

Division method in hashing, 725

Documentation

- comments, 768-770

- declaration order, 771-773

- formatting lines and expressions, 773-774

- identifiers, 770-771

- indentation, 774-775

- specifications as, 8

Domains, functional, 49

Dot (.) and dot notation

- with access variables, 195

- for ranges, 798

- with records, 82-83, 243, 800

Doubly linked lists, 397-398

- deleting elements from, 402

- with dummy nodes, 398, 403-404

- finding elements in, 398-400

- inserting elements in, 400-401

- for queues, 287

- for unbounded natural numbers, 413-414

Draw procedure (Bingo_Basket), 118, 135-137

Drivers, 46

- for Binary_Search procedure, 63-66

- in bottom-up testing, 53-55

Dummy nodes

- doubly linked lists with, 398, 403-404

- linked lists with, 376-386

Dummy values in nonlinked binary search trees, 598

Duplicate keys, 409-410

- in binary search trees, 559

- in circular linked lists, 393

- in linked linear lists, 349

- in sequential lists, 335

Duration type in Standard, 767

Dynamic memory allocation

- for recursion, 487, 490-496

- for stacks, 241-251

- of strings, 193-195.

- See also* Access type variables

E

Easy_Test program, 45

EBNF definitions, 740

- for compilation, 784

- for generic units, 146

- list of, 740-751

- for package body, 18, 321

- for package specifications, 17-18

Edge_Rec record, 632

Edge_Type record, 619

Edges for graphs, 615-617

Efficiency

- of algorithms, 21-26

- versus complexity, 26

- of implementations, 252

- in recursion, 500-501

- in sorting, 23, 654, 685-687

- of traverse operations, 326

Efficient programs, 4

Elaboration, 491, 811

Electronic address books, 314-326

Element function (Bounded_Strings), 178, 188, 190

Element function (Unbounded_Strings), 204, 211

else clauses, 804

elsif clauses, 804

'EMAX' attribute, 777

Employee_Rec record, 690

Empty function (Binary_Search_Tree), 540

Empty function (Priority_Queue), 610, 614

Empty function (Queue), 270

- array-based, 283

- linked list-based, 289

Empty function (Sorted_List), 311

- for array-based linked list implementation, 447-448

- for file-based sorted list implementation, 455, 462

- for linked linear list implementation, 357

- for sequential list implementation, 342

- Empty function (Stack), 224, 237, 247, 250

Empty operations

- for FIFO queues, 269, 279

- for linked linear lists, 343

- for linked lists with dummy nodes, 379

- for sequential lists, 329

Empty sets, 131, 138

Empty stacks, 223

Encapsulation, 99-102

- benefits of, 253-254

- for information hiding, 16

- private types for, 20

- end if statements, 804

Enqueue operations

- for FIFO queues, 269-270, 279-280, 285-287

- in priority queues, 608, 612

Enqueue procedure (`Priority_Queue`), 609, 613

Enqueue procedure (`Queue`), 270

- array-based, 282

- linked list-based, 288-289

Entry calls, attribute for, 777

`ENUMERATION_IO` package, 758, 809-810

Enumeration types, input/output of, 758, 809-810

'`EPSILON`' attribute, 777

`Equal` function (`Address_Book`), 323

`Equal` function (`Bounded_Strings`), 177, 187

`Equal` function (`Client`), 550

`Equal` function (=) (`Unbounded_Strings`), 203, 209

`Equal` sign (=)

- for arrays, 90

- precedence of, 803

Equality

- of arrays, 802

- of records, 165, 800

Error checking for array elements, 92

Errors

- compile-time, 35-36

- meaningful messages for, 225

- run-time, 36-37

- in specification and design, 33-35

Escape program

- design of, 506-512

- implementation of, 513-517

- specification for, 504-505, 512-513

- testing, 517-518

`Evaluate` function (`Postfix`), 232-233

Example program, 795-798

Exceptions and exception handlers, 36, 811-812

- with access type variables, 198

- in `Direct_IO`, 761

- and postconditions, 225-226

- in `Sequential_IO`, 759

- for stacks, 225-226, 238, 247

- in `Standard`, 767

- in `Text_IO`, 758

Exclusive-ORs in folding hash method, 727-728

Execution time

- of algorithms, 21

- variable binding at, 487

Exhaustive testing, 49-50

`exit` statement, 805

Explicit type conversions, 456

Exponent values, attributes for, 777, 780-781

Exponential time, 24

Expressions

- formatting, 773-774

- postfix, 226-234

F

Factorial function

- iteration for, 475

- recursion for, 470-474

- run-time stack for, 493-496

Fibonacci function, 481

Fibonacci numbers, 479-481

Field selectors, 82

Fields, record, 82, 243

FIFO queues

- abstract level of, 268-271

- application level of, 271-277

- arrays for, 278-283

- circular, 280-281, 290

- for graph searches, 626-628

- implementation level of, 278-294

- linked structures for, 284-290

- operations on, 269-271, 279

- performance of, 283, 290-294

- testing, 294

File-based sorted lists

- abstract level of, 454-455

- implementation level of, 455-464

`File_Type` type, 16

Files, 320

- binary, 810

- `Direct_IO` routines for, 760

- linked lists as, 452-464

- memory management functions for, 457

- `Sequential_IO` routines for, 759

- text, 809-810

- `Text_IO` routines for, 754

`Finalize` procedure (`Unbounded_Natural`), 412-413, 424

`Finalize` procedure (`Unbounded_Strings`), 205, 210

`Find_And_Unlink_Max` procedure, 566-568, 576-577

`Find_Average` procedure (`Auto_Part_List`), 409

`Find_Minimum` function, 170-171

`Find` procedure (`Client`), 546-547, 551

Finding. *See* Searching

Finite collections, 88

'`FIRST`' attribute, 777-778, 799-800, 802

'`FIRST_BIT`' attribute, 778

'`FIRST(N)`' attribute, 778

`FIXED_IO` package, 757

Fixed-length strings, 176

`Flight_Rec` record, 630

`Float_Array_Ops` package, 153

`FLOAT_IO` package, 756-757

Float type, 16, 799

- as atomic class, 80

- formal generic type parameter for, 151

- in `Standard`, 764-765

Folding hash method, 727-730

'`FORE`' attribute, 778

Formal parameters, 143, 145-152, 806

Formatting programs

- comments in, 768-770

- declaration order in, 771-773

- identifiers in, 770-771

- indentation in, 774-775

- lines and expressions, 773-774

FORTRAN language, features in, 35

`Fraction_Class` package, 53-55

`Fraction` function (`Discrete_Random`), 793

`Free_Node` function (`Sorted_List`), 440, 445, 448-450, 459

`Free` procedure, 197, 201-202

Free-space stacks, 449

Freight train manifests application, 272-277

`Friend_Rec` record, 317-318

Full binary trees, 597

`Full` function (`Binary_Search_Tree`), 540

`Full` function (`Priority_Queue`), 610, 613-614

`Full` function (`Queue`), 270

- array-based, 283

- linked list-based, 289

`Full` function (`Sorted_List`), 311

- for array-based linked list implementation, 447

- for file-based sorted list implementation, 455, 462

- for linked linear list implementation, 356-357

- for sequential list implementation, 341

`Full` function (`Stack`), 224, 237-238, 247, 250

Full operations

- for FIFO queues, 270-271, 279, 285

- for linked linear lists, 343

- for sequential lists, 329

Function decomposition, 13-14

Functional abstraction, 14

Functional correctness, 20

Functional domains, 49

Functional subproblems, 13

Functionally correct designs, 279

Functions

- comments for, 770

- identifiers for, 771

- in-line, 686-687

- indenting in, 772-773

- operators as, 807-808

- returning values from, 807

G

Games, Bingo simulation, 117-123

Garbage collection, 196

General cases

 in recursion, 476-477

 testing, 50

Generic formal objects, 179

Generic formal parameters, 143, 145

Generic formal private types, 154-155, 166

Generic formal subprograms, 155-157

Generic formal types, 146-154

Generic packages, 790-791, 809-810

Generic units, 142-146

Get_Adjacent_Vertices procedure (Graph), 621, 635, 637

Get_Card procedure (Bingo_Card_ADO), 103, 105

Get_Card procedure (Bingo_Card_ADT), 109-110, 112

Get_Cards procedure (Bingo_Simulation), 122

Get_Delegate_Data procedure (Translations), 171

Get_Item procedure (Postfix), 232

Get_Line procedure (Bounded_Strings), 179, 189

Get_Line procedure (Unbounded_Strings), 205, 212

Get_List procedure, 64

Get_List procedure (Example), 797

Get_Maze procedure (Escape), 515

Get procedure (Unbounded_Natural), 412, 415, 423-424

Get_Start_Point procedure (Escape), 515

Glass box testing, 50

Global variables, 108

Goals

 in software engineering, 3-6

 for testing, 49-52

Graph package, 619-622, 635-640

Graph theory, 616

Graph_Type record, 633

Graphs

 abstract level of, 615-622

 application level of, 622-632

 array-based implementations of, 632-638

 breadth-first searching in, 626-629

 depth-first searching in, 622-626

 implementation level of, 632-640

 linked lists for, 638-640

 single-source shortest paths problem in, 629-632

 specification for, 619-622

 traversing, 622-632

Greater-than (>) function (Bingo_Number_Set), 133, 141

Greater-than (>) function (Bounded_Strings), 177, 186

Greater-than (>) function (Discrete_Set), 145, 149

Greater-than (>) function (Unbounded_Strings), 203, 209

Greater-than operator (>) for arrays, 90, 802

Greater-than-or-equal-to (>=) function (Bingo_Number_Set), 133, 141

Greater-than-or-equal-to (>=) function (Bounded_Strings), 177, 187

Greater-than-or-equal-to (>=) function (Discrete_Set), 145, 149

Greater-than-or-equal-to (>=) function (Heap_Sort), 684

Greater-than-or-equal-to (>=) function (Unbounded_Strings), 203, 209

Greater-than-or-equal-to operator (>=) for arrays, 90

Greater-than sign (>), precedence of, 803

Greedy algorithms, 169

Group programming, 16-17, 43, 53

H

Hardware in software engineering, 3

Hash functions, 714, 726-727, 729

Hashing, 713-716

 collisions in, 716-724

 function selection for, 724-730

Header comments, 768-769

Header nodes, lists with, 376-379

 deleting elements in, 385-386

 doubly linked, 403-404

 finding elements in, 380-385

 inserting elements into, 385-386

 traversing, 379-380

Heap, 215

Heap package, 602-603, 606-608

Heap_Sort procedure, 683-684

Heap sorts, 680-685

Heaps, 598-602

 building, 680-682

 implementing, 603-608

 versus priority queues, 614-615

 specification for, 602-603

Height in binary trees, 535-537, 562

Hiding information, 12-13

 advantages in, 81-82

 encapsulation for, 16

Hierarchies in abstraction, 11

High-probability ordering, 710

Homogeneous collections, 308

Homogeneous elements, 88

Hump yard application, 272-277

Hump_Yard procedure, 275

Hyphen (–)

 for comments, 798

 precedence of, 803

Hypotheses in debugging, 57

I

Ideaware, 3

Identifiers, 43, 770-772, 798

Identity function, 624

Identity function (Heap_Sort), 684

Identity function (Part_List), 312-313

if statements, 804

'IMAGE' attribute, 778

Implementation level

 of binary search trees, 554-578

 of classes, 80

 of data structures, 97-99

 of FIFO queues, 278-294

 of file-based sorted lists, 455-464

 of graphs, 632-640

 of linear lists, 326-361

 of one-dimensional arrays, 90-93

 of priority queues, 611-614

 of records, 84-85

 of sets, 137-142

 of stacks, 235-254

 of strings, 185-192

 of two-dimensional arrays, 95

 of unbounded-length strings, 208-216

Implementations, 16-20

 comparing, 251-253

 of unbounded natural numbers, 414-424

In-line comments, 769-770, 772

In-line subprograms, 686-687

In-order traversals, 541-542

in out parameter mode, 496, 806

in parameter mode, 496, 806

Incomplete type declarations, 249

Indentation, 772-775

Independently testable units, 45

Index function (Bounded_Strings), 178, 188-190

Index function (Unbounded_Strings), 204, 211

Index_Of function procedure (Graph), 634-635

Indexes

 for arrays, 88-89, 92-93, 158-163, 801

 attributes for, 779, 781

 with FIFO queues, 279-280

 for strings, 176

Inductive proofs, 476

Inequality

 of arrays, 802

 of records, 800

Infinite loops, 40-41

Infix notation, 226

Information hiding, 12-13

- advantages in, 81-82

- encapsulation for, 16

Initialize_Memory procedure (Sorted_List), 444, 450

Initializing

- abstract data objects, 321

- access types and record types, 377

- circular list searches, 390-391

- record elements, 796, 801

INLINE pragma, 686-687

Inorder procedure, 568

Input

- exception handlers for, 36

- in specifications, 7-8

- of text files, 809-810

Input routines

- in Direct_IO, 760

- in Sequential_IO, 759

- in Text_IO, 754

Insert operations

- for binary search trees, 557-562, 572-574

- in Bounded_Strings, 179, 189, 204-205, 212

- for circular linked lists, 392-395

- for doubly linked lists, 400-401

- for linked linear lists, 348-352

- for linked lists with dummy nodes, 385-386

- for sequential lists, 334-336

Insert procedure

- for binary search trees, 559, 574

- for circular linked lists, 394-395

- for doubly linked lists, 401

- with hashing, 715-718

- for linked lists with dummy nodes, 385-386

Insert procedure (Binary_Search_Tree), 539

Insert procedure (Sorted_List), 310

- for array-based linked list implementation, 446

- for file-based sorted list implementation, 455, 460-461

- for linked linear list implementation, 355

- for sequential list implementation, 340

Insertion_Location function, 666

Insertion_Sort procedure, 666

Insertion sorts, 665-667

Inspections, 43-44

Instances for generic units, 142

Instantiations

- declarations for, 772

- in package specifications, 18

- of packages, 143, 790-791, 795-796, 809-810

INTEGER_IO package, 756

Integer_Ops package, 149-150

Integer type, 16, 799

- as atomic class, 80

- formal generic type parameter for, 149-150

- levels of, 80-81

- in Standard, 763-764

- Text_IO routines for, 756

Interactive input, exception handlers for, 36

Interactive test drivers, 63-64

Interfaces

- package body specifications as, 18

- problems from, 39

Interpolation searches, 713

Intersection of sets, 131-132

Interval procedure (Client), 547, 552

Invalid postfix expressions, 230-231

Invariants, loop, 39-42, 332

Inventory records

- advanced iterators for, 406-409

- as sorted list, 312-314

Is_Member function (Bingo_Number_Set), 132, 140

Is_Member function (Discrete_Set), 144, 147

Iterative operations

- on binary search trees, 570-578

- versus recursion, 475, 497-499

Iterators, 308-309

- advanced, 406-409

- for Part_List, 313-314

- for Sorted_List, 311

K

KEY_ERROR exceptions, 334, 338, 388, 562

Key_List record, 619

Key-ordered lists, 308

Keyboard input, exception handlers for, 36

Keys

- duplicate, 335, 349, 393, 409-410, 559

- for hashing, 725-726

- ordering by, 308, 710-711

- in records, 308

- for sorts, 688-689

"Kind of" hierarchies, 11

L

Labels procedure (Address_Book), 320, 325

Language Reference Manual (LRM), 795

Language translations case study, 167-172

Languages, features in, 35-36

'LARGE attribute, 779

Large-scale problems, object-oriented design for, 14

Larger_Child function (Heap), 607

Largest safe numbers, attribute for, 781

'LAST attribute, 779, 799-800, 802

'LAST_BIT attribute, 779

Last in, first out (LIFO) structures. *See* Stacks

'LAST (N) attribute, 779

Last_Name_First procedure (Sort_Names)

- with bounded-length strings, 182-183

- with unbounded-length strings, 207-208

Laundry problem, Big-O analysis of, 24-25

Leaf nodes in binary trees, 535

Left children in binary trees, 534-536

Left expression in set operations, 142

Left subtrees, 535

Length

- of identifiers, 771

- of strings, 176, 190

- Text_IO routines for, 755

'LENGTH attribute, 779, 800, 802

Length function (Bounded_Strings), 178, 188, 190

Length function (Sorted_List), 311

- for array-based linked list implementation, 448

- for file-based sorted list implementation, 455, 457, 462

- for linked linear list implementation, 357

- for sequential list implementation, 342

Length function (Unbounded_Natural), 416

Length function (Unbounded_Strings), 203, 211

'LENGTH (N) attribute, 779

Length operations for linked linear lists, 344

Less-than (<) function (Address_Book), 323

Less-than (<) function (Bingo_Number_Set), 133, 141

Less-than (<) function (Bounded_Strings), 177, 186

Less-than (<) function (Client), 550

Less-than (<) function (Discrete_Set), 145, 148-149

Less-than (<) function (Unbounded_Strings), 203, 209

Less-than operator (<) for arrays, 90, 802

Less-than-or-equal-to (<=) function (Bingo_Number_Set), 133, 141

Less-than-or-equal-to ($\leq$) function (`Bounded_Strings`), 177, 186

Less-than-or-equal-to ($\leq$) function (`Discrete_Set`), 145, 148

Less-than-or-equal-to ($\leq$) function in sorts, 689

Less-than-or-equal-to ($\leq$) function (`Unbounded_Strings`), 203, 209

Less-than-or-equal-to operators ($\leq$) for arrays, 90

Less-than sign ($<$), precedence of, 803

Levels in binary trees, 535-536, 562

Libraries

for generic units, 143

program, 785-787

Library units

EBNF definition of, 784

versus secondary units, 784-785

Life-cycle verification activities, 58

Limited private types, 111, 155

for encapsulation, 253-254

for unbounded-length stings, 202

Line control routines in `Text_IO`, 755

Line length routines in `Text_IO`, 755

Line numbers in testing, 44

Linear lists

abstract level of, 308-314

application level of, 314-326

with dummy nodes, 376-386

implementation level of, 326-361

linked. *See* Linked linear lists

performance of, 358-361

sequential. *See* Sequential linear lists

testing, 361-362

Linear probing, 717-720

`Linear_Search` procedure, 709

`Linear_Search2` procedure, 711

Linear searching, 708-711

Linear time, 23

Lines (program), formatting, 773-774

Lines (queues). *See* FIFO queues

Link time, variable binding at, 487

Linked linear lists, 342-344

deleting elements in, 352

with dummy nodes, 376-386

finding elements in, 344-348

inserting elements into, 348-352

modifying elements in, 348

performance of, 358-361

retrieving elements in, 348

specification for, 353-357

testing, 361-362

Linked lists

as array of records, 438-452

circular, 386-397

doubly linked, 397-404

with dummy nodes. *See* Linked lists with dummy nodes

as file of records, 452-464

for graphs, 638-640

linear. *See* Linked linear lists

memory allocation for, 327

performance of, 405-406

recursive processing of, 483-485

for unbounded natural numbers, 413-414

Linked lists with dummy nodes

deleting elements in, 385-386

empty operations for, 379

finding elements in, 380-385

inserting elements into, 385-386

traversing, 379-380

Linked structures,

See also Binary search trees

for linear lists, 327-328, 342-361

for queues, 284-290

for stacks, 241-253

`List_Type` record, 86, 328, 377, 655, 708, 717

Lister program, specifications for, 8-9

Lists

arrays for, 83-84

with duplicate elements, 409-410

linear. *See* Linear lists; Linked linear lists; Sequential linear lists

linked. *See* Linked linear lists; Linked lists; Linked lists with dummy nodes

operations for, 308-309

`Load` procedure (`Bingo_Basket`), 118, 135-136

Local global variables, 108

Local variables

access variables as, 205

activation records for, 491

memory for, 215

Logarithmic time, 23, 60

Logical errors, 37

Logical operators

precedence of, 803

in `Standard`, 763

`Lookup` procedure (`Address_Book`), 319, 324

Loop invariants, 39-42, 332

Loops, 805

in algorithms, 21

checking, 43

infinite, 40-41

with recursion, 475

Loose coupling of modules, 13

Lower bounds

for arrays, 90-91

attributes for, 777-778

Lowercase characters in `Standard`, 766

M

'`MACHINE_EMAX`' attribute, 780

'`MACHINE_EMIN`' attribute, 780

'`MACHINE_MANTISSA`' attribute, 780

'`MACHINE_OVERFLOWS`' attribute, 780

'`MACHINE_RADIX`' attribute, 780

'`MACHINE_ROUNDS`' attribute, 780

Magic numbers, 43

Magnitude, Big-O notation for, 22-26

`Mailing_Label_Rec` record, 688

Maintenance phase

changes in, 5

verification activities in, 58

Manifests application, 272-277

'`MANTISSA`' attribute, 780

`Mark` procedure (`Maze`), 506, 513-514

`Mark_Vertex` procedure (`Graph`), 621, 624, 635, 638

`Marked` function (`Graph`), 621-622, 635, 638

Matrices, adjacency, 632-634

`Max_Size` object, 440

`Maximum` function (`Integer_Ops`), 149-150

Maximum heaps, 599

Maze case study, 504-518

Maze package, 512-514

Measurements for testing progress, 51-52

Measures for algorithms, 21

Median function, 811

Medical emergency response application

implementation of, 548-554

specification for, 543-548

Members, set, 130

Memory and memory management

for arrays, 90-92, 327, 439-440, 448-450

binding, with variables, 487

dynamic storage allocation in, 490-496

for files, 457

for linked lists, 327

organization of, 215

for records, 84-85

static storage allocation in, 487-490

for strings, 193-195

Memory leaks, 196-198, 201-202, 205

`Merge` function, 669, 671

`Merge_Sort` procedure, 671

Merge sorts, 668-673

Metric-based testing, 51-52

Midtest loops, 805

`Min_Index` function (`Selection_Sort`), 657-659

Minimizing translations case study, 167-172

`Minimum` function (`Integer_Ops`), 149-150

Minimum heaps, 599

Minimum height in binary trees, 535-537

Minus sign (–)

- for comments, 798

- precedence of, 803

Minus sign (–) function (`Bingo_Number_Set`), 133, 140-141

Minus sign (–) function (`Discrete_Set`), 144, 148

Minus sign (–) function (`Float_Array_Ops`), 153-154

Minus sign (–) function (`Fraction_Class`), 53

Model numbers, attributes for, 777, 779, 782

Models, 10

Modes for parameters, 806-807

Modifiable programs, 5

Modifications in modular design, 26

Modify operations

- for binary search trees, 556-557, 577

- for linked linear lists, 348

- for sequential lists, 334-335

Modify procedure (`Binary_Search_Tree`), 539-540

Modify procedure (`Sorted_List`), 310

- for array-based linked list implementation, 447

- for file-based sorted list implementation, 455-456, 461

- for linked linear list implementation, 356

- for sequential list implementation, 341

Modula-2 language, features in, 35

Multidimensional arrays, 93-95

Multiple lines for statements, 773

Multiplication

- in postfix expressions, 227

- precedence of, 803

Multiply procedure (`Unbounded_Natural`), 412, 415, 418, 422

N

`NAME_ERROR` exceptions, 326

`Name_Of` function (`Address_Book`), 323

`Name_Of` function (`Client`), 550

`Name_Of` function (`Part_List`), 312

`Name_Rec` record, 318, 546, 688

`Name_Tree` package, 549

Named associations

- with arrays, 803

- with parameters, 806

- with records, 800-801

Named constants, 18

Named numbers, declarations for, 772

Names

- alphabetizing, 180-184

- for arrays, 90

- for files, 454

- for identifiers, 43

- for objects, 770-771

- overloading, 134

- in records, 317

- scope and visibility of, 798

- for source files, 784

Natural numbers, unbounded

- design for, 413-414

- implementation of, 414-424

- specification for, 411-413

Natural type, 80, 799

Naur, Peter, 740

Negation operator in `Standard`, 763

Nesting of subprograms, 408, 798

`new` allocator, 194-195

`New_Node` function (`Sorted_List`), 439

- for array-based linked list implementation, 444, 448-449

- for file-based sorted list implementation, 457-459

`Node_Type` record, 249, 284, 343, 377, 483, 554

Nodes, 242-243

- in binary trees, 534-536

- dummy, 376-386

- for graphs, 615-617

- in heaps, 599-601

- in nonlinked binary search trees, 596-597

Nonlinked binary search trees, 596-598

Not equals operator (`/=`)

- for arrays, 90, 802

- precedence of, 803

- for records, 800

`not` operator

- with arrays, 139

- precedence of, 803

Nouns for identifiers, 771

`null` access value, 194, 198, 377

Null addresses with arrays, 439

`null` pointers

- for circular linear lists, 388

- for linked linear lists, 342

Null ranges, 798

Numbers

- converting, 728-729, 803-804

- declarations for, 772

- input and output of, 810

- natural, unbounded, 411-424

- random, 136, 722

O

Object code

- memory for, 488

- in program libraries, 785

Object-oriented design, 14-16

Objects, 14-15

- abstract, 102-108, 321

- access variables for, 193

- director, 119

- generic formal, 179

- identifiers for, 770-771

- for strings, 176

Observers, 308, 311

Obsolete units, 790

Offsets

- for array elements, 92

- for record addresses, 85

- storage unit, 778-779, 781

On-line debugging programs, 56

One-dimensional arrays

- abstract level of, 88-90

- application level for, 90

- declaration of, 796

- implementation level for, 90-93

Operands in postfix expressions, 226-231

Operating systems, 3

Operational scenarios, 7

Operations

- for arrays, 88, 90, 802

- for classes, 15

- as functions, 807-808

- precedence of, 803

- for private types, 111, 254

Operator functions, 134-135

Operators

- in postfix expressions, 226-231

- in `Standard`, 763-765

- for strings, creating, 184-185

- style for, 773-774

Optimal solutions, 168

`or else` operators, 331

`or` operator

- with arrays, 139

- precedence of, 803

Order

- of compilation, 787-790

- of declarations, 771-773

- in sorts, 687-688

Order property for heaps, 598, 601

Ordered elements, 88

Ordering.

- See also* Sorts

- high-probability, 710

- key, 710-711

Orders of magnitude, 22-26

`others` clause, 804

`out` parameter mode, 496, 806

Output

- in specifications, 7-8

- of text files, 809-810

Output routines

- in `Bounded_Strings`, 179, 189, 204-205, 212

- in `Direct_IO`, 760

- in `Sequential_IO`, 759

- in `Text_IO`, 754

`Oven_Class` package, 18-20

Overflow

- attribute for, 780
- in postfix expressions, 231
- queue, 271
- stack, 225, 238, 247

Overloading, 134, 807

P

P_Queue package, 630

Package body, EBNF definition of, 321

Packages

- for classes, 18-20
- comments for, 770
- declarations for, 772
- implementing, 16-20
- instantiations of, 143, 790-791, 795-796, 809-810
- syntax of, 17-18
- testing, 46

Page control routines in Text_IO, 755

Page length routines in Text_IO, 755

Parameters

- access type, 202
- activation records for, 496
- arrays as, 808
- association of, 806
- attribute for, 781
- binding, 487-488
- checking, 43
- with discriminants, 86
- formal, 143, 145-152
- memory for, 215
- modes for, 806-807
- in overloading, 807
- program libraries for tracking, 785
- stacks for, 234
- style for, 773
- undefined, 48

Parent nodes, 597

Parentheses ()

- in operator precedence, 803
- for two-dimensional arrays, 93

Parents in binary trees, 534-536

Part_List package, 312-314

"Part of" hierarchies, 11

Part_Rec record, 312, 378, 406

Part_Sort procedure, 156

Pascal versus Ada, 795-798

- array types, 801-803
- comments, 798
- declarations, 798
- elaborations, 811
- exceptions, 811-812
- identifiers, 798
- input/output, 809-810
- numeric type conversion, 803-804
- operator precedence, 803
- record types, 800-801
- statements, 804-805
- subprograms, 806-808
- types and subtypes, 798-800

Passive error checking, 36

Paths

- in graphs, 617
- testing, 50-51

Performance

- of array-based linked list implementations, 450-452
- of binary search tree implementations, 578-580
- of bubble sorts, 662-664
- of Escape program, 518-519
- of FIFO queues, 283, 290-294
- of file-based linked sorted implementations, 464
- of heap sorts, 684-685
- of insertion sorts, 667
- of linear and binary searches, 712-713
- of linear list implementations, 358-361
- of linked list implementations, 405-406
- of merge sorts, 672-673
- of priority queues, 614-615
- of quick sorts, 679-680
- of radix sorts, 696-697
- of recursion, 500-501
- of selection sorts, 659-660
- of stacks, 251-253

Period (.)

- with access variables, 195
- for ranges, 798
- with records, 82-83, 243, 800

Phone numbers in records, 317

Phone_Of function (Client), 550

Phone_Rec record, 318, 546

Phone_Tree package, 549

Pivot values in quick sorts, 675-678

Placeholders, dummy nodes as, 376

Plans

- for debugging, 56-57
- for testing, 52, 66-67

Play_All_Games procedure (Bingo_Simulation), 122-123

Play_One_Game procedure (Bingo_Simulation), 121

Plus sign (+), precedence of, 803

Plus sign (+) function (Bingo_Number_Set), 132-133, 140

Plus sign (+) function (Client), 551

Plus sign (+) function for complex numbers, 808

Plus sign (+) function (Discrete_Set), 144, 147-148

Plus sign (+) function (Float_Array_Ops), 153-154

Plus sign (+) function (Fraction_Class), 53

Plus sign (+) operator for sets, 132

Pointers.

- See also* Access type variables for lists. *See* Linked linear lists; Linked lists; Linked lists with dummy nodes
- sorting, 689-691

Pop procedure (Stack), 225, 238-240, 244-248, 251

Popping stack elements, 223, 246-248

'POS attribute, 781, 800

Position

- of parameters, attribute for, 781
- in radix sorts, 692-697

'POSITION attribute, 781

Positional associations

- with arrays, 803
- with parameters, 806
- with records, 800

Positive type, 80, 799

Postconditions

- checking, 43
- comments for, 101
- in design, 38-39
- and exceptions, 225-226

Postfix expressions, 226-230

- invalid, 230-231
- procedure for, 231-234

Posttest loops, 805

Pragmas, 686-687

Precedence

- of operators, 803
- in postfix expressions, 226

Preconditions

- checking, 43
- in design, 38-39

'PRED attribute, 781, 799

Pred function

- attribute for, 781
- in binary searches, 161

Predefined attributes, 776-783

Predefined library units, 786

Prefix procedure (Unbounded_Natural), 417-418

Preinstantiation of generic packages, 790-791

Preorder traversals, 542-543

Pretest loops, 39, 805

Primary keys for sorts, 688-689

Prime numbers in rehashing, 721-722

Print_List procedure, 64-65

Print_Manifest procedure (Hump_Yard), 276

Print_One_Label procedure (Address_Book), 323-324

Printers, queues for, 271-272

Priority_Queue package, 609-610, 613-614

Priority queues

- abstract level of, 608-610

- application level of, 610-611
- for graph searches, 629-632
- verses heaps, 614-615
- implementation level of, 611-614

Private types

- for encapsulation, 20, 253-254
- generic formal, 154-155, 166
- operations with, 111

Problems, defining, 6-7

Procedural abstraction, 14

Procedures

- comments for, 770
- identifiers for, 771
- in-line, 686-687
- indenting in, 772-773
- nested, 408
- testing, 46

Process_One_Part procedure (Auto_Part_List), 409

Processing requirements in specifications, 7-9

Programming style

- comments, 768-770
- declaration order, 771-773
- formatting lines and expressions, 773-774
- identifiers, 770-771
- indentation, 774-775

Programs

- compilation units in, 784-785
- designing. *See* Design
- libraries for, 785-787
- testing, 44-49
- verifying, 32

Proper subsets, 131

Pseudorandom number generators, 136, 722, 792-794

Psychological distance in identifiers, 770-771

Push procedure (Stack), 224, 237-239, 241, 244-245, 251

Pushing stack elements, 223

Put_All_Negatives procedure (Part_List), 314

Put_Array procedure (Example), 797

Put calls

- for branch testing, 51
- for debugging, 57, 69

Put_If_Negative procedure (Part_List), 314

Put procedure (Bounded_Strings), 179, 189

Put procedure (Client), 550-551

Put procedure (Unbounded_Natural), 412, 424

Put procedure (Unbounded_Strings), 205, 212

Put_Set procedure, 170

Q

Quadratic probing, 722

Quadratic time, 23

Qualification of names, 809

Queue package, 270-271

- array-based, 282-283
- linked list-based, 288-289

Queue_Type record

- for arrays, 278, 282
- for linked structures, 284

Queues

- FIFO. *See* FIFO queues
- priority. *See* Priority queues
- in radix sorts, 692-697

Quick fixes, 57

Quick_Sort procedure, 678-679

Quick sorts, 674-680

Quotation marks (") for operator symbols, 134

R

Radix, attribute for, 780

Radix_Sort procedure, 694-696

Radix sorts, 691-697

Railroad yard application, 272-277

Random access files, 452

Random function (Discrete_Random), 793-794

Random number generators, 136, 722, 792-794

Random probing, 722

'RANGE' attribute, 781, 800, 802

'RANGE (N)' attribute, 781

Ranges, 89, 798-799

Read_Node procedure (Sorted_List), 456, 458

Read procedure (Direct_IO), 452, 456

Readability, 4-5

- of names, 770-771
- spaces for, 773-774

Real types, Text_IO routines for, 756-757

Reclaiming memory, 196-198, 201-202, 205

Records, 800-801

- abstract level of, 82-83
- activation, 490-492
- application level of, 83-84
- arrays of, linked lists as, 438-452
- for bounded-length strings, 185-186
- discriminants, 85-88
- equality of, 165
- file of, linked lists as, 452-464
- implementation level of, 84-85
- initializing, 377, 796
- key values in, 308
- sorting, 688-689
- for stacks, 235, 242

Recursion

- in binary searches, 485-487, 555-569
- boundary problems in, 511
- considerations in, 500-503
- debugging routines using, 497
- for factorials, 470-475
- for linked lists, 483-485
- in maze case study, 504-518
- with multiple recursive calls, 479-481
- operation of, 487-496
- removing, 497-500
- simplification from, 481-483
- tail, 498-499
- verifying procedures using, 475-476
- writing procedures using, 476-481

Recycling memory, 196-198, 201-202, 205

Reference, passing parameters by, 496

Regression testing, 55-56

Rehashing, 720-722

Reheap Down operation, 601

Reheap_Down procedure (Heap), 603-605, 607

Reheap Up operation, 601

Reheap_Up procedure (Heap), 603-608

Relational operators

- for arrays, 90
- in Bounded_Strings, 177, 186-187, 189-190
- precedence of, 803
- for sets, 133, 144-145, 148-149
- in Standard, 763
- in Unbounded_Strings, 202-203, 209

Relative addresses in dynamic storage allocation, 490

Relatively prime numbers, 721-722

Reliability, 26, 254

rem operator

- for circular arrays, 281
- in hashing, 725
- precedence of, 803

Remove_Extra procedure, 184-185

Remove_Zeros procedure (Unbounded_Natural), 417

Removing

- blanks from strings, 184-185
- set elements, 131-132

Reports, inspection, 44

Requirements

- Processing, 7-9
- program, 4

Reserved words

- capitalizing, 771
- list of, 739

Resources in specifications and packages, 17

Restore procedure (Client), 548, 553

Retrieve operations

for binary search trees, 556-557, 577

for linked linear lists, 348

for sequential lists, 334

Retrieve procedure

for binary search trees, 556-557, 577

with hashing, 715, 718-719

Retrieve procedure (Binary_Search_Tree), 540

Retrieve procedure (Graph), 620, 634, 636-637

Retrieve procedure (Sorted_List), 310

for array-based linked list implementation, 447

for file-based sorted list implementation, 455, 462

for linked linear list implementation, 356

for sequential list implementation, 341

Return addresses, stacks for, 234

return statement, 807

Reuse of code, 15, 59, 116

array attributes for, 802

in Binary_Search procedure, 157-166

determining, 320

generic units for, 142-146

reverse statement, 805

Reverse_Traverse procedure

nonrecursive, 499-500

recursive, 484-485

Right children in binary trees, 534-536

Right expression in set operations, 142

Right subtrees, 535

Robustness, 36

Root nodes in heaps, 599-601

Roots in binary trees, 534-536

Rounding, attribute for, 780

Row_Winner function (Bingo_Card_ADO), 106

Row_Winner function (Bingo_Card_ADT), 113-114

Rows in tables, 93

Run-time errors, 36-37

Run-time stacks, 492-496

S

'SAFE_EMAX attribute, 781

'SAFE_LARGE attribute, 781

'SAFE_SMALL attribute, 781

Save procedure (Address_Book), 320-325

Saving access variables, 438

Scalar types, 80-82

Scenarios, operational, 7

Scope, 408, 798

Search_Array procedure, 330

Search_Array procedure (Sorted_List), 339-340

Search_Circular_List procedure

for circular linked lists, 389-392

for doubly linked lists, 399-400

Search_Linked_List procedure

for linked linear lists, 345-348

for linked lists with dummy nodes, 380-385

Search_Linked_List procedure (Sorted_List)

for array-based list implementation, 443, 445

for file-based list implementation, 459-460

for linked linear list implementation, 354

Search_Tree procedure, 570-572

Search trees. *See* Binary search trees

Searching, 59

binary, 59-70, 712-713

binary search trees, 570-572

circular linked lists, 389-392

doubly linked lists, 398-400

efficiency of, 23

in graphs, 622-632

hashing. *See* Hashing

linear, 708-711

linked linear lists, 344-348

linked lists with dummy nodes, 380-385

reusable procedures for, 157-166

sequential linear lists, 330-334

Secondary units

compilation of, 787

EBNF definition of, 784

versus library units, 784-785

Selection_Sort procedure, 156, 658

Selection_Sort procedure (Sort_Names)

for bounded-length strings, 182

for unbounded-length strings, 206-207

Selection sorts, 655-660

Selectors, field, 82

Semantic errors, 35

Sequential access files, 452, 810

Sequential_IO package, 320, 759-760, 810

Sequential linear lists, 327-329

deleting elements from, 336-338

finding elements in, 330-334

inserting elements in, 334-336

modifying elements in, 334-335

performance of, 358-361

retrieving elements in, 334

specification for, 339-342

traversing, 338-339

Sequential searches, 709

of arrays, 330-334

Big-O notation for, 59

Set notation for vertices and edges, 616

Set_Seed procedure (Discrete_Random), 792-793

Sets

abstract level of, 130-135

application level of, 135-137

implementation level of, 137-142

operations on, 131-132, 139

specifications for, 132-135

translations case study using, 167-172

Shape

of binary trees, 562-563

of heaps, 598, 601-602

Short_Bubble_Up procedure, 663-664

Short-circuit evaluation of Boolean expressions, 331

Shortest_Paths procedure, 630-632

Side effects from aliases, 197-198

Sidebar comments, 770

Simulations, Bingo, 117-123

Single quotes (') for attributes, 799

Single-source shortest paths problem, 629-632

Single stepping, 56, 687

'SIZE attribute, 782

Size function (Binary_Search_Tree), 540

Size function (Direct_IO), 453

Size operations for binary search trees, 555-556

Skewed binary trees, 562-563

Skewed keys in hashing, 725-726

Slash (/), precedence of, 803

Slash (/) function (Float_Array_Ops), 153-154

Slash (/) function (Fraction_Class), 53

Slice function (Bounded_Strings), 178, 188

Slice procedure (Unbounded_Strings), 203-204, 211

Slices

of arrays, 89-90, 92, 802

of record items, 82-83

of strings, 176

'SMALL attribute, 782

Smaller-caller question in recursion, 476

Smaller_Name function, 691

Smallest safe numbers, attribute for, 781

Software and software engineering, 2-3

- algorithm comparisons, 20-26

- goals for, 3-6

- implementation of, 16-20

- program design, 10-16

- specifications in, 6-10

- toolbox for, 3

Software development tools, 56

Solutions, optimal and greedy, 168-169

Some_Sort procedure, 654-655

Sort_Names procedure

- with bounded-length strings, 181-184

- with unbounded-length strings, 205-208

Sort_Lip procedure, 689-690

Sorted_List package

- for array-based linked list implementation, 441-448

- for file-based sorted list implementation, 454-464

- for linked linear list implementation, 353-357

- for sequential list implementation, 339-342

- specification for, 309-314

Sorted lists, 308

Sorts

- array, 155-156

- bubble, 660-664

- comparing, 697-698

- efficiency of, 23, 654, 685-687

- heap, 680-685

- insertion, 665-667

- keys for, 688-689

- merge, 668-673

- for names, 180-184

- order in, 687-688

- for pointers, 689-691

- quick sort, 674-680

- radix, 691-697

- selection, 655-660

- specification for, 654-655

Source files, names for, 784

Spaces in expressions, 773-774

Special cases, testing, 50

Specifications, 4,6-10

- for Address Book, 315-316

- for binary search trees, 538-541

- for Bingo card, 100-108

- for bounded-length strings, 177-180, 186-192

- for Calendar, 762

- for Direct_IO, 760-761

- errors in, 33-35

- for Escape program, 504-505, 512-513

- for graphs, 619-622

- for heaps, 602-603

- for medical emergency response application, 544-548

- for packages, 17-18

- for Sequential_IO, 759-760

- for sets, 132-135

- for sorts, 654-655

- for Standard, 763-767

- for Text_IO, 754-758

- for unbounded natural numbers, 411-413

Split procedure, 675-679

Split values in quick sorts, 675-679

Square brackets ([]) in EBNF definitions, 740

Stack_Array type, 235

Stack package, 224-225

- array-based, 236-240

- link-based, 250-253

Stack_Type type, 235, 249

Stacks

- abstract level of, 222-226

- for activation records, 492-496

- application level of, 226-234

- arrays for, 235-240

- for Escape program, 509

- exception handling for, 225-226, 238, 247

- free-space, 449

- for graph searches, 623-624

- implementation level of, 235-254

- linked structures for, 241-253

- in memory, 215

- operations on, 222-225

- performance of, 251-253

- versus recursion, 499-500

Standard package, 176, 763-767

Statements, 804-805

- breaking, 773

- indentation for, 774-775

Static arrays for queues, 278-283

Static storage allocation, 487-490

Stepping, 55, 687

Storage allocation

- dynamic, 193-200, 241-251, 487, 490-496

- static, 487-490

STORAGE_ERROR exceptions, 215

- with access type variables, 198

- with recursion, 497

- in Text_IO, 756

'STORAGE_SIZE' attributes, 782

Storage unit offsets, attribute for, 778-779, 781

Storage units reserved, attributes for, 782

Strings, 16, 200-202, 799, 801-802

- abstract level for, 176-180, 202-205

- access types for, 193-200

- access variables with, 198-200

- in Ada 83, 191-192

- in Ada 95, 215-216

- alphabetizing, 180-184

- application level for, 180-185, 205-208

- comparing, 189-190

- creating operators for, 184-185

- dynamic memory allocation of, 193-200

- in hashing, 726-730

- implementation level for, 185-192, 208-216

- records for, 185-186

- removing blanks from, 184-185

- sorting, 180-184, 205-208

- specification for, 177-180, 186-192

- in Standard, 767

- substring operations, 184

- terminology for, 176

Structured integration testing, 52-56

Structures, 95-96

- abstract level of, 96-97

- access variables with, 198-200

- application level of, 96

- implementation level of, 97-99

- as parameters, 496

Stubs, 55, 65-66

Student_Rec record, 409

Style, programming

- comments, 768-770

- declaration order, 771-773

- formatting lines and expressions, 773-774

- identifiers, 770-771

- indentation, 774-775

Subprograms, 806-808

- activation records for, 492

- generic formal, 155-157

- in-line, 686-687

- as library units, 785

- nesting of, 798

- overloading names of, 134

- in package specifications, 18

- recursive, 473-474

- stacks for, 234

- testing, 53

Subsets, 131

Substring operations, 184, 190-191

Subtract procedure (Unbounded_Natural), 412, 420-422

Subtraction

- in postfix expressions, 227

- of unbounded natural numbers, 415

Subtrees, 535-537

Subtypes, 798-800

- array, 796

- in package specifications, 18

- in Standard, 766

'SUCC attribute, 782, 799-800

Succ function

- attribute for, 782

- in binary searches, 161

Swap procedure (Bubble_Sort), 662

Swap procedure (Heap), 606-607

Swap procedure (Heap_Sort), 684

Swap procedure (Integer_Ops), 149-150

Swap procedure (Quick_Sort), 678

Swap procedure (Selection_Sort), 658

Swap procedure (Short_Bubble_Sort), 663

Swap_Values procedure, 154-155

Symbol_At function (Maze), 506-507, 513-514

Symbol tables, 488, 785

Symptoms in debugging, 57

Syntax

- EBNF definition list for, 740-751

- errors in, 35-36

- of packages, 17-18

T

Table size for hashing, 724-725

Tables, 93-95

Tail recursion, 498-499

Task execution, attribute for, 776

Team programming, 16-17, 43, 53

Templates

- classes as, 15

- for data structures, 130

'TERMINATED attribute, 782

Terminated tasks, attribute for, 782

Terminating conditions for loops, 40-41

Test_Add procedure (Unbounded_Natural), 411

Test_Divider procedure, 46-48

Test_Driver procedure, 64

Test drivers, 46

- for Binary_Search procedure, 63-66

- in bottom-up testing, 53-55

Test_Fractions procedure, 54-55

Testing, 32.

- See also* Debugging

- bottom-up, 53-55

- code coverage in, 50-52

- data coverage in, 49-50

- deskchecking, 42-43

- Escape program, 517-518

- FIFO queues, 294

- goals for, 49-52

- linear lists, 361-362

- plans for, 52, 66-67

- practical considerations in, 56

- programs, 44-49

- recursion, 475-476

- regression, 55-56

- set membership, 131

- structured integration, 52-56

- top-down, 55

- units, 45-46, 143

- verification activities in, 37, 58

- walkthroughs and inspections for, 43-44

Testing phase, changes in, 5

Text editors, 3

Text files, 320, 809-810

Text_IO package, 16, 754-758, 809

Three-question method, 476

Time

- constant and linear, 23

- in software engineering, 5-6

Timing tests, 21

- of binary search tree implementations, 580

- of FIFO queues, 293

- of linear and binary searches, 713

- of linear list implementations, 360-361

- of linked list implementations, 406, 451

- of recursion implementations, 500-501

- of sorts, 698

- of stacks, 253

Title_Of function, 611

To_Bit_String function, 728

To_Bounded_String function (Bounded_Strings), 179, 189

To_Integer function (Postfix), 232

To_Natural function, 729

To_String function (Bounded_Strings), 179, 189

To_String function (Unbounded_Strings), 204, 212

To_String132 function, 180

To_Unbounded_String procedure (Unbounded_Strings), 204, 212

Top-down design, 13-14

Top-down testing, 55

Top of stacks, 222

Tracking program execution, 42-43, 56-57

Trailer nodes, lists with, 376-379

- deleting elements in, 385-386

- doubly linked, 403-404

- finding elements in, 380-385

- inserting elements into, 385-386

- traversing, 379-380

Train manifests application, 272-277

Translations case study, 167-172

Translations procedure, 169-170

Traversal_Order type, 539

Traverse operations

- for binary search trees, 541-543, 568-569

- for circular linked lists, 388-389

- efficiency of, 326

- for graphs, 622-632

- for linked linear lists, 344

- for linked lists with dummy nodes, 379-380

- recursion for, 483-485

- for sequential lists, 338-339

Traverse procedure

- for circular linked lists, 389

- for linked lists with dummy nodes, 380

Traverse procedure (Binary_Search_Tree), 541

Traverse procedure (Part_List), 313

Traverse procedure (Sorted_List), 312

- for array-based linked list implementation, 448

- for file-based sorted list implementation, 455, 462-463

- for linked linear list implementation, 357

- for sequential list implementation, 342

Try_To_Escape procedure (Escape), 507-510, 516-518

Two-dimensional arrays

- abstract level of, 93-95

- application level of, 95

- declaration of, 796

- implementation level of, 95

Types, 798-800

- abstract, 108-116

- for array indexes, 159-161

- as classes, 80

- composite, *See* Composite types

- converting, 456, 803-804

- declarations for, 249, 772, 795

- discriminants for, 85-88

- generic formal, 146-154

- in generic formal parameters, 143

- generic formal private, 154-155

- identifiers for, 771

- in package specifications, 18

- private, 20, 111, 154-155, 166, 253-254

- scalar, 80-82

U

Unbounded-length strings, 176, 200-202

- abstract level of, 202-205

- access types for, 193-200

- application level of, 205-208

- implementation level of, 208-216

Unbounded natural numbers

- design for, 413-414
- implementation of, 414-424
- specification for, 411-413

Unbounded_Natural package

- implementation of, 416-424
- specification for, 411-413

Unbounded-Strings package

- implementation of, 208-212
- specification for, 202-205

Unchecked_Deallocation procedure 196-198

Unconstrained arrays, 801-802

- declaration of, 796
- as parameters, 808

Unconstrained types

- for array indexes, 159
- for arrays, 86

Undefined parameters, 48

Underflow

- in postfix expressions, 231
- queue, 271
- stack, 225, 238, 247

Undirected graphs, 616-618

Union of sets, 131-132, 134, 139

Unique keys, 308

Units

- compilation, 784-785
- generic, 142-146
- obsolete, 790
- testing, 45-46

Universal sets, 131, 138

Unordered collections, 130

Upper bounds

- for arrays, 90-91
- attributes for, 779

Usable programs, 4

use clauses, 134-135, 809

User/operational scenarios, 7

V

'VAL attribute, 782-783, 800

Valid_Number function (Bingo_Card_ADO), 104-105

Valid_Number function (Bingo_Card_ADT), 112

'VALUE attribute, 783

'Value_In_Array function

- using iteration, 497-499
- using recursion, 477-479

Values

- atomic, 80
- in graphs, 618
- ordering lists by, 308

Variables

- access type. *See* Access type variables
- activation records for, 491
- for array indexes, 89
- binding memory with, 487
- in debugging, 56-57
- declarations for, 772, 796-797
- for discriminants, 86-87
- file, 453
- global, 108
- identifiers for, 771
- memory for, 215
- for records, 82-83

Varying-length strings, 176

Verbs for identifiers, 771

Verification, *See* Testing

Vertex_Rec record, 632

Vertical bar (|)

- in ceiling function, 535
- in EBNF definitions, 740

Vertices in graphs, 615-617

Visibility of names, 798

W

Waiting lines. *See* FIFO queues

Walkthroughs, 43-44

Weight_Of function (Graph), 621, 634, 637

Weighted graphs, 618-622

'WIDTH attribute, 783

Winner function (Bingo_Card_ADO), 103, 108

Winner function (Bingo_Card_ADT), 110-111, 115

with clauses, 17-18, 784-785, 809

Work_Request_Rec record, 610

Write_Node procedure (Sorted_List), 456, 458

Write_One_Friend procedure (Address_Book), 324

Write procedure (Direct_IO), 452, 456

X

xor operator, precedence of, 803

Z

Zero, division by, 36