LEWIS | CHASE

java™ SOFTWARE STRUCTURES

Third Edition

Designing and Using Data Structures

java™ Third Edition
SOFTWARE STRUCTURES
Designing and Using Data Structures

This page intentionally left blank

java™ SOFTWARE STRUCTURES
Third Edition

Designing and Using Data Structures

JOHN LEWIS
Virginia Tech

JOSEPH CHASE
Radford University

Addison-Wesley
New York Boston San Francisco
London Toronto Sydney Tokyo Singapore Madrid
Mexico City Munich Paris Cape Town Hong Kong Montreal

Editor-in-Chief	Michael Hirsch
Editorial Assistant	Stephanie Sellinger
Managing Editor	Jeffrey Holcomb
Production Supervisor	Heather McNally
Marketing Manager	Erin Davis
Marketing Coordinator	Kathryn Ferranti
Senior Manufacturing Buyer	Carol Melville
Online Product Manager	Bethany Tidd
Art Director	Linda Knowles
Cover Design	Elena Sidorova
Project Management, Composition, and Illustrations	Nesbitt Graphics, Inc.
Project Coordinator, Nesbitt Graphics, Inc.	Harry Druding
Project Manager, Nesbitt Graphics, Inc.	Kathy Smith
Text Design, Nesbitt Graphics, Inc.	Jerilyn Bockorick, Alisha Webber
Cover Image	Steve Cole/Getty Images

Library of Congress Cataloging-in-Publication Data

Lewis, John, 1963-
  Java software structures : designing and using data structures / John
Lewis, Joseph Chase. -- 3rd ed.
    p. cm.
  Includes bibliographical references and index.
  ISBN 978-0-13-607858-6 (alk. paper)
 1. Java (Computer program language) 2. Data structures (Computer science)
3. Computer software--Development. I. Chase, Joseph. II. Title.
  QA76.73.J38L493 2009
  005.13'3--dc22

2009000302

Addison Wesley
is an imprint of

PEARSON

ISBN-13: 978-013-607858-6

ISBN-10: 0-13-607858-3

www.pearsonhighered.com

1 2 3 4 5 6 7 8 10–EB–13 12 11 10 09

To my wife Sharon and my kids:
Justin, Kayla, Nathan, and Samantha

–J. L.

To my loving wife Melissa for her support and encouragement
and to our families, friends, colleagues, and students who have provided
so much support and inspiration through the years.

–J. C.

This page intentionally left blank

Preface

This book is designed to serve as a text for a course on data structures and algorithms. This course is typically referred to as the CS2 course because it is often taken as the second course in a computing curriculum. We have designed this book to embrace the tenets of Computing Curricula 2001 (CC2001).

Pedagogically, this book follows the style and approach of the leading CS1 book *Java Software Solutions: Foundations of Program Design*, by John Lewis and William Loftus. Our book uses many of the highly regarded features of that book, such as the Key Concept boxes and complete code examples. Together, these two books support a solid and consistent approach to either a two-course or three-course introductory sequence for computing students. That said, this book does not assume that students have used *Java Software Solutions* in a previous course.

Material that might be presented in either course (such as recursion or sorting) is presented in this book as well. We also include strong reference material providing an overview of object-oriented concepts and how they are realized in Java.

We understand the crucial role that the data structures and algorithms course plays in a curriculum and we think this book serves the needs of that course well.

The Third Edition

We have made some key modifications in this third edition to enhance its pedagogy. The most important change is a fundamental reorganization of material that is designed to create a cleaner flow of topics. Instead of having an early, large chapter to review object-oriented concepts, we've included that material as an appendix for reference. Then we review concepts as needed and appropriate in the context of the implementation strategies discussed throughout the book and cite the appropriate reference material. This not only links the topics in a timely fashion but also demonstrates the usefulness of particular language constructs.

We've expanded the discussion of Analysis of Algorithms, and given it its own chapter. The discussion, however, stays at an appropriately moderate level. Our strategy is to motivate the concepts involved in the analysis of algorithms, laying a solid foundation, rather than get embroiled in too much formality.

Another key organizational change is that the introduction to collections uses a stack as the primary example. In previous editions of this book we went out of our way to introduce collections in an abstract way that separated it from the core data structures, using examples such as a bag or set collection. This new approach capitalizes on the fact that a stack is conceptually about as straightforward as it gets. Using it as a first example enhances the understanding of collections as a whole.

The previous edition of the book had several chapters that focused on larger case studies that made use of collections to solve non-trivial problems. While many instructors found these useful, they also seemed to interrupt the flow of coverage of core topics. Therefore we have taken the case study chapters out of the book and put them on the web as supplementary resources. We encourage all instructors to download and use these resources as they see fit.

Finally, for this edition we've reviewed and improved the discussions throughout the book. We've expanded the discussion of graphs and reversed the order of the graphs and hashing chapters to make a cleaner flow. And we've added a chapter that specifically covers sets and maps.

We think these modifications build upon the strong pedagogy established by previous editions and give instructors more opportunity and flexibility to cover topics as they choose.

Our Approach

Books of this type vary greatly in their overall approach. Our approach is founded on a few important principles that we fervently embraced. First, we present the various collections explored in the book in a consistent manner. Second, we emphasize the importance of sound software design techniques. Third, we organized the book to support and reinforce the big picture: the study of data structures and algorithms. Let's examine these principles further.

Consistent Presentation

When exploring a particular type of collection, we carefully address each of the following issues in order:

1. **Concept:** We discuss the collection conceptually, establishing the services it provides (its interface).
2. **Use:** We explore examples that illustrate how the particular nature of the collection, no matter how it's implemented, can be useful when solving problems.
3. **Implementation:** We explore various implementation options for the collection.
4. **Analysis:** We compare and contrast the implementations.

The Java Collections API is included in the discussion as appropriate. If there is support for a particular collection type in the API, we discuss it and its implementation. Thus we embrace the API, but are not completely tied to it. And we are not hesitant to point out its shortcomings.

The analysis is kept at a high level. We establish the concept of Big-Oh notation in Chapter 2 and use it throughout the book, but the analysis is more intuitive than it is mathematical.

Sound Program Design

Throughout the book, we keep sound software engineering practices a high priority. Our design of collection implementations and the programs that use them follow consistent and appropriate standards.

Of primary importance is the separation of a collection's interface from its underlying implementation. The services that a collection provides are always formally defined in a Java interface. The interface name is used as the type designation of the collection whenever appropriate to reinforce the collection as an abstraction.

In addition to practicing solid design principles, we stress them in the discussion throughout the text. We attempt to teach both by example and by continual reinforcement.

Clean Organization

The contents of the book have been carefully organized to minimize distracting tangents and to reinforce the overall purpose of the book. The organization supports the book in its role as a pedagogical exploration of data structures and algorithms as well as its role as a valuable reference.

The book can be divided into numerous parts: Part I consists of the first two chapters and provides an introduction to the concept of a collection and analysis of algorithms. Part II includes the next four chapters, which cover introductory and underlying issues that affect all aspects of data structures and algorithms as well as linear collections (stacks, queues, and lists). Part III covers the concepts of recursion, sorting, and searching. Part IV covers the nonlinear collections (trees, heaps, hashing, and graphs). Each type of collection, with the exception of trees, is covered in its own chapter. Trees are covered in a series of chapters that explore their various aspects and purposes.

Chapter Breakdown

Chapter 1 (**Introduction**) discusses various aspects of software quality and provides an overview of software development issues. It is designed to establish the appropriate mindset before embarking on the details of data structure and algorithm design.

Chapter 2 (**Analysis of Algorithms**) lays the foundation for determining the efficiency of an algorithm and explains the important criteria that allow a developer to compare one algorithm to another in proper ways. Our emphasis in this chapter is understanding the important concepts more than getting mired in heavy math or formality.

Chapter 3 (**Collections**) establishes the concept of a collection, stressing the need to separate the interface from the implementation. It also conceptually introduces a stack, then explores an array-based implementation of a stack.

Chapter 4 (**Linked Structures**) discusses the use of references to create linked data structures. It explores the basic issues regarding the management of linked lists, and then defines an alternative implementation of a stack (introduced in Chapter 3) using an underlying linked data structure.

Chapter 5 (**Queues**) explores the concept and implementation of a first-in, first-out queue. Radix sort is discussed as an example of using queues effectively. The implementation options covered include an underlying linked list as well as both fixed and circular arrays.

Chapter 6 (**Lists**) covers three types of lists: ordered, unordered, and indexed. These three types of lists are compared and contrasted, with discussion of the operations that they share and those that are unique to each type. Inheritance is used appropriately in the design of the various types of lists, which are implemented using both array-based and linked representations.

Chapter 7 (**Recursion**) is a general introduction to the concept of recursion and how recursive solutions can be elegant. It explores the implementation details of recursion and discusses the basic idea of analyzing recursive algorithms.

Chapter 8 (**Sorting and Searching**) discusses the linear and binary search algorithms, as well as the algorithms for several sorts: selection sort, insertion sort, bubble sort, quick sort, and merge sort. Programming issues related to searching and sorting, such as using the Comparable interface as the basis of comparing objects, are stressed in this chapter. Searching and sorting that are based in particular data structures (such as heap sort) are covered in the appropriate chapter later in the book.

Chapter 9 (**Trees**) provides an overview of trees, establishing key terminology and concepts. It discusses various implementation approaches and uses a binary tree to represent and evaluate an arithmetic expression.

Chapter 10 (**Binary Search Trees**) builds off of the basic concepts established in Chapter 9 to define a classic binary search tree. A linked implementation of a binary search tree is examined, followed by a discussion of how the balance in the tree nodes is key to its performance. That leads to exploring AVL and red/black implementations of binary search trees.

Chapter 11 (**Priority Queues and Heaps**) explores the concept, use, and implementations of heaps and specifically their relationship to priority queues. A heap sort is used as an example of its usefulness as well. Both linked and array-based implementations are explored.

Chapter 12 (**Multi-way Search Trees**) is a natural extension of the discussion of the previous chapters. The concepts of 2-3 trees, 2-4 trees, and general B-trees are examined and implementation options are discussed.

Chapter 13 (**Graphs**) explores the concept of undirected and directed graphs and establishes important terminology. It examines several common graph algorithms and discusses implementation options, including adjacency matrices.

Chapter 14 (**Hashing**) covers the concept of hashing and related issues, such as hash functions and collisions. Various Java Collections API options for hashing are discussed.

Chapter 15 (**Sets and Maps**) explores these two types of collections and their importance to the Java Collections API.

Appendix A (**UML**) provides an introduction to the Unified Modeling Language as a reference. UML is the de facto standard notation for representing object-oriented systems.

Appendix B (**Object-Oriented Design**) is a reference for anyone needing a review of fundamental object-oriented concepts and how they are accomplished in Java. Included are the concepts of abstraction, classes, encapsulation, inheritance, and polymorphism, as well as many related Java language constructs such as interfaces.

Supplements

The following supplements are available to all readers of this book at www.aw.com/cssupport.

- **Source Code** for all programs presented in the book
- **Full case studies** of programs that illustrate concepts from the text, including a Black Jack Game, a Calculator, a Family Tree Program, and a Web Crawler

The following instructor supplements are only available to qualified instructors at Pearson Education's Instructor Resource Center, http://www.pearsonhighered.com/irc. Please visit the Web site, contact your local Pearson Education Sales Representative, or send an e-mail to computing@pearson.com, for information about how to access them.

- **Solutions** for selected exercises and programming projects in the book
- **Test Bank,** containing questions that can be used for exams
- **PowerPoint® Slides** for the presentation of the book content

Acknowledgements

First and most importantly we want to thank our students for whom this book is written and without whom it never could have been. Your feedback helps us become better educators and writers. Please continue to keep us on our toes.

We would like to thank all of the reviewers listed below who took the time to share their insight on the content and presentation of the material in this book and its previous editions. Your input was invaluable.

Mary P. Boelk,	Marquette University
Robert Burton,	Brigham Young University
Gerald Cohen,	St. Joseph's College
Robert Cohen,	University of Massachusetts–Boston
Jack Davis,	Radford University
Bob Holloway,	University of Wisconsin–Madison
Nisar Hundewale,	Georgia State University
Chung Lee,	California State Polytechnic University
Mark C. Lewis,	Trinity University
Mark J. Llewellyn,	University of Central Florida
Ronald Marsh,	University of North Dakota
Eli C. Minkoff,	Bates College; University of Maine–Augusta
Ned Okie,	Radford University
Manuel A. Perez-Quinones,	Virginia Tech
Moshe Rosenfeld	University of Washington
Salam Salloum,	California State Polytechnic University–Pomona
Don Slater,	Carnegie Mellon University
Ashish Soni,	University of Southern California
Carola Wenk,	University of Texas–San Antonio

The folks at Addison-Wesley have gone to great lengths to support and develop this book along with us. It is a true team effort. Editor-in-Chief Michael Hirsch and his assistant Stephanie Sellinger have always been there to help. Marketing Manager Erin Davis, her assistant Kathryn Ferranti, and the entire Addison-Wesley sales force work tirelessly to make sure that instructors understand the goals and benefits of the book. Heather McNally flawlessly handled the production of the book, and Elena Sidorova is to be credited for the wonderful cover design. They are supported by Kathy Smith and Harry Druding at Nesbitt Graphics. Carol Melville always finds a way to get us time on press so

that our book makes it into your hands in time to use it in class. Thank you all very much for all your hard work and dedication to this book.

We'd be remiss if we didn't acknowledge the wonderful contributions of the ACM Special Interest Group on Computer Science Education. Its publications and conferences are crucial to anyone who takes the pedagogy of computing seriously. If you're not part of this group, you're missing out.

Finally, we want to thank our families, who support and encourage us in whatever projects we find ourselves diving into. Ultimately, you are the reason we do what we do.

This page intentionally left blank

Contents

1 Introduction

Our exploration of data structures begins with an overview of the underlying issues surrounding the quality of software systems. It is important for us to understand that it is necessary for systems to work as specified, but simply working is not sufficient. We must also develop quality systems. This chapter discusses a variety of issues related to software quality and data structures, and it establishes some terminology that is crucial to our exploration of data structures and software design.

CHAPTER OBJECTIVES

- Identify various aspects of software quality
- Motivate the need for data structures based upon quality issues
- Introduce the basic concept of a data structure
- Introduce several elementary data structures

1.1 Software Quality

Imagine a scenario where you are approaching a bridge that has recently been built over a large river. As you approach, you see a sign informing you that the bridge was designed and built by local construction workers and that engineers were not involved in the project. Would you continue across the bridge? Would it make a difference if the sign informed you that the bridge was designed by engineers and built by construction workers?

The word "engineer" in this context refers to an individual who has been educated in the history, theory, method, and practice of the engineering discipline. This definition includes fields of study such as electrical engineering, mechanical engineering, and chemical engineering. *Software engineering* is the study of the techniques and theory that underlie the development of high-quality software.

When the term "software engineering" was first coined in the 1970s, it was an aspiration—a goal set out by leaders in the industry who realized that much of the software being created was of poor quality. They wanted developers to move away from the simplistic idea of writing programs and toward the disciplined idea of engineering software. To engineer software we must first realize that this term is more than just a title and that it, in fact, represents a completely different attitude.

Many arguments have been started over the question of whether software engineering has reached the state of a true engineering discipline. We will leave that argument for software engineering courses. For our purposes, it is sufficient to understand that as software developers we share a common history, we are constrained by common theory, and we must understand current methods and practices in order to work together.

Ultimately, we want to satisfy the *client*, the person or organization who pays for the software to be developed, as well as the final *users* of the system, which may include the client, depending on the situation.

The goals of software engineering are much the same as those for other engineering disciplines:

- Solve the right problem
- Deliver a solution on time and within budget
- Deliver a high-quality solution
- Accomplish these things in an ethical manner (see www.acm.org/about/code-of-ethics)

Sir Isaac Newton is credited with the quote "If I have seen further it is by standing on the shoulders of giants." As modern software developers, we stand upon the shoulders of the giants that founded and developed our field. To truly stand upon their shoulders, we must understand, among other things, the fundamentals of quality software systems and the organization, storage, and retrieval of data. These are the building blocks upon which modern software development is built.

Quality Characteristic	Description
Correctness	The degree to which software adheres to its specific requirements.
Reliability	The frequency and criticality of software failure.
Robustness	The degree to which erroneous situations are handled gracefully.
Usability	The ease with which users can learn and execute tasks within the software.
Maintainability	The ease with which changes can be made to the software.
Reusability	The ease with which software components can be reused in the development of other software systems.
Portability	The ease with which software components can be used in multiple computer environments.
Efficiency	The degree to which the software fulfills its purpose without wasting resources.

FIGURE 1.1 Aspects of software quality

To maximize the quality of our software, we must first realize that quality means different things to different people. And there are a variety of quality characteristics to consider. Figure 1.1 lists several aspects of high-quality software.

Correctness

The concept of *correctness* goes back to our original goal to develop the appropriate solution. At each step along the way of developing a program, we want to make sure that we are addressing the problem as defined by the requirements specification and that we are providing an accurate solution to that problem. Almost all other aspects of quality are meaningless if the software doesn't solve the right problem.

Reliability

KEY CONCEPT
Reliable software seldom fails and, when it does, it minimizes the effects of that failure.

If you have ever attempted to access your bank account electronically and been unable to do so, or if you have ever lost all of your work because of a failure of the software or hardware you were using, you are already familiar with the concept of *reliability*. A software *failure* can be defined as any unacceptable behavior that occurs within permissible operating conditions. We can compute measures of reliability, such as the mean time between failures or the number of operations or tasks between failures.

In some situations, reliability is an issue of life and death. In the early 1980s, a piece of medical equipment called the Therac-25 was designed to deliver a dose of radiation according to the settings made by a technician on a special keyboard. An error existed in the software that controlled the device and when the technician made a very specific adjustment to the values on the keyboard, the internal settings of the device were changed drastically and a lethal dose of radiation was issued. The error occurred so infrequently that several people died before the source of the problem was determined.

In other cases, reliability repercussions are financial. On a particular day in November, 1998, the entire AT&T network infrastructure in the eastern United States failed, causing major interruption in communications capabilities. The problem was eventually traced back to a specific software error. That one failure cost millions of dollars in lost revenue to the companies affected.

Robustness

Reliability is related to how *robust* a system is. A robust system handles problems gracefully. For example, if a particular input field is designed to handle numeric data, what happens when alphabetic information is entered? The program could be allowed to terminate abnormally because of the resulting error. However, a more robust solution would be to design the system to acknowledge and handle the situation transparently or with an appropriate error message.

Developing a thoroughly robust system may or may not be worth the development cost. In some cases, it may be perfectly acceptable for a program to abnormally terminate if very unusual conditions occur. On the other hand, if adding such protections is not excessively costly, it is simply considered to be good development practice. Furthermore, well-defined system requirements should carefully spell out the situations in which robust error handling is required.

Usability

To be effective, a software system must be truly *usable*. If a system is too difficult to use, it doesn't matter if it provides wonderful functionality. Within the discipline of computer science there is a field of study called Human-Computer Interaction (HCI) that focuses on the analysis and design of user interfaces of software systems. The interaction between the user and system must be well designed, including such things as help options, meaningful messages, consistent layout, appropriate use of color, error prevention, and error recovery.

Maintainability

KEY CONCEPT
Software systems must be carefully designed, written, and documented to support the work of developers, maintainers, and users.

Software developers must *maintain* their software. That is, they must make changes to software in order to fix errors, to enhance the functionality of the system, or simply to keep up with evolving requirements. A useful software system may need to be maintained for many years after its original development. The software engineers who perform maintenance tasks are often not the same ones as those who originally developed the software. Thus, it is important that a software system be well structured, well written, and well documented in order to maximize its maintainability.

Large software systems are rarely written by a single individual or even a small group of developers. Instead, large teams, often working from widely distributed locations, work together to develop systems. For this reason, communication among developers is critical. Therefore, creating maintainable software is beneficial for the long term as well as for the initial development effort.

Reusability

Suppose that you are a contractor involved in the construction of an office building. It is possible that you might design and build each door in the building from scratch. This would require a great deal of engineering and construction effort, not to mention money. Another option is to use pre-engineered, prefabricated doors for the doorways in the building. This approach represents a great savings of time and money because you can rely on a proven design that has been used many times before. You can be confident that it has been thoroughly tested and that you know its capabilities. However, this does not exclude the possibility that a few doors in the building will be custom engineered and custom built to fit a specific need.

When developing a software system, it often makes sense to use pre-existing software components if they fit the needs of the developer and the client. Why reinvent the wheel? Pre-existing components can range in scope from the entire system, to entire subsystems, to individual classes and methods. They may come from part of another system developed earlier or from libraries of components that are created to support the development of future systems. Some pre-existing components are referred to as Commercial Off-The-Shelf (COTS) products. Pre-existing components are often reliable because they have been tested in other systems.

Using pre-existing components can reduce the development effort. However, reuse comes at a price. The developer must take the time to investigate potential components to find the right one. Often the component must be modified or extended to fit the criteria of the new system. Thus, it is helpful if the component is truly *reusable*. That is, software should be written and documented so that it can be easily incorporated into new systems and easily modified or extended to accommodate new requirements.

Portability

Software that is easily *portable*, can be moved from one computing environment to another with little or no effort. Software developed using a particular operating system and underlying central processing unit (CPU) may not run well or at all in another environment. One obvious problem is a program that has been compiled into a particular CPU's machine language. Because each type of CPU has its own machine language, porting it to another machine would require another, translated version. Differences in the various translations may cause the "same" program on two types of machines to behave differently.

Using the Java programming language addresses this issue because Java sourcecode is compiled into *bytecode*, which is a low-level language that is not the machine language for any particular CPU. Bytecode runs on a *Java Virtual Machine* (JVM), which is software that interprets the bytecode and executes it. Therefore, at least theoretically, any system that has a JVM can execute any Java program.

Efficiency

The last software quality characteristic listed in Figure 1.1 is efficiency. Software systems should make *efficient* use of the resources allocated to them. Two key resources are CPU time and memory. User demands on computers and their software have risen steadily ever since computers were first created. Software must always make the best use of its resources in order to meet those demands. The efficiency of individual algorithms is an important part of this issue and is discussed in more detail in the next chapter and throughout the book.

KEY CONCEPT
Software must make efficient use of resources such as CPU time and memory.

Quality Issues

To a certain extent, quality is in the eye of the beholder. That is, some quality characteristics are more important to certain people than to others. We must consider the needs of the various *stakeholders*, the people affected one way or another by the project. For example, the end user certainly wants to maximize reliability, usability, and efficiency, but doesn't necessarily care about the software's maintainability or reusability. The client wants to make sure the user is satisfied, but is also worried about the overall cost. The developers and maintainers want the internal system quality to be high.

KEY CONCEPT
Quality characteristics must be prioritized, and then maximized to the extent possible.

Note also that some quality characteristics are in competition with each other. For example, to make a program more efficient, we may choose to use a complex algorithm that is difficult to understand and therefore hard to maintain. These types of trade-offs require us to carefully prioritize the issues related to a particular project and, within those boundaries, maximize all quality characteristics as much as possible. If we decide that we must use the more complex algorithm for efficiency, we can also document the code especially well to assist with future maintenance tasks.

Although all of these quality characteristics are important, for our exploration of data structures in this book, we will focus upon reliability, robustness, reusability and efficiency. In the creation of data structures, we are not creating applications or end-user systems but rather reusable components that may be used in a variety of systems. Thus, usability is not an issue since there will not be a user interface component of our data structures. By implementing in Java and adhering to Javadoc standards, we also address the issues of portability and maintainability.

1.2 Data Structures

Why spend so much time talking about software engineering and software quality in a text that focuses on data structures and their algorithms? Well, as you begin to develop more complex programs, it's important to develop a more mature outlook on the process. As we discussed at the beginning of this chapter, the goal should be to engineer software, not just write code. The data structures we examine in this book lay the foundation for complex software that must be carefully designed. Let's consider an example that will illustrate the need for and the various approaches to data structures.

A Physical Example

Imagine that you must design the process for how to handle shipping containers being unloaded from cargo ships at a dock. In a perfect scenario, the trains and trucks that will haul these containers to their destinations will be waiting as the ship is unloaded and thus there would not be a need to store the containers at all. However, such timing is unlikely, and would not necessarily provide the most efficient result for the trucks and the trains since they would have to sit and wait while each ship was unloaded. Instead, as each shipping container is unloaded from a ship, it is moved to a storage location where it will be held until it is loaded on either a truck or a train. Our goal should be to make both the unloading of containers and the retrieval of containers for transportation as efficient as possible. This will mean minimizing the amount of searching required in either process and minimizing the number of times a container is moved.

Before we go any further, let's examine the initial assumptions underlying this problem. First, our shipping containers are generic. By that we mean that they are all the same shape, same size, and can hold the same volume and types of materials. Second, each shipping container has a unique identification number. This number is the key that determines the final destination of each container and whether it is to be shipped by truck or train. Third, the dock workers handling the containers do not need to know, nor can they know, what is inside of each container.

Given these assumptions as a starting point, how might we design our storage process for these containers? One possibility might be to simply lay out a very large array indexed by the identification number of the container. Keep in mind, however, the identification number is unique, not just for a single ship but for all ships and all shipping containers. That means that we would need to layout an array of storage at each dock large enough to handle all of the shipping containers in the world. This would also mean that at any given point in time, the vast majority of the units in these physical arrays would be empty. This would appear to be a terrible waste of space and very inefficient. We will explore issues surrounding the efficiency of algorithms in the next chapter.

We could try this same solution but allow the array to collapse and expand like an `ArrayList` as containers are removed or added. However, given the physical nature of this array, such a solution may involve having to move containers multiple times as other containers with lower ID numbers are added or removed. Again, such a solution would be very inefficient.

What if we could estimate the maximum number of shipping containers that we would be storing on our dock at any given point in time? This would create the possibility of creating an array of that maximum size and then simply placing each container into the next available position in the array as it is unloaded from the ship. This would be relatively efficient for unloading purposes since the dock workers would simply keep track of which locations in the array were empty and place each new container in an empty location. However, using this approach, what would happen when a truck driver arrived looking for a particular container? Since the array is not ordered by container ID, the driver would have to search for the container in the array. In the worst case scenario, the container they are seeking would be the last one in the array causing them to search the entire array. Of course, the average case would be that each driver would have to search half of the array. Again, this seems less efficient than it could be.

Before we try a new design, let's reconsider what we know about each container. Of course, we have the container ID. But from that, we also have the destination of each container. What if instead of an abitrary ordering of containers in a one-dimensional array, we create a two-dimensional array where the first dimension is the destination? Thus we can implement our previous solution as the second dimension within each destination. Unloading a container is still quite simple. The dock workers would simply take the container to the array for its destination and place it in the first available empty slot. Then our truck drivers would not have to search the entire dock for their shipping container, only that portion, the array within the array, that is bound for their destination. This solution, which is beginning to behave like a data structure called a *hash table* (discussed in Chapter 13), is an improvement but still requires searching at least a portion of the storage.

The first part of this solution, organizing the first dimension by destination, seems like a good idea. However, using a simple unordered array for the second dimension still does not accomplish our goal. There is one additional piece of information that we have for each container that we have yet to examine—the order that it has been unloaded from the ship. Let's consider an example for a moment of the components of an oil rig being shipped in multiple containers. Does the order of those containers matter? Yes. The crew constructing the oil rig must receive and install the foundation and base components before being able to construct the top of the rig. Now the order the containers are removed from the ship and the order they are placed in storage is important. To this point, we have been considering only the problem of unloading, storing, and shipping storage containers. Now we begin to see that this problem exists within a larger context including how the ship was loaded at its point of origin. Three possibilities exist for how the ship was loaded: containers that are order dependent were loaded in the order they are needed, containers that are order dependent were loaded in the reverse order they are needed, or containers that are order dependent were loaded without regard to order but with order included as part of the information associated with the container ID. Let's consider each of these cases separately.

Keep in mind that the process of unloading the ship will reverse the order of the loading process onto the ship. This behavior is very much like that of a *stack*, data structure, which we will discuss further in Chapters 3 and 4. If the ship was loaded in the order the components are needed, then the unloading process will reverse the order. Thus our storage and retrieval process must reverse the order again in order to get the containers back into the correct order. This would be accomplished by taking the containers to the array for their destination and storing them in the order they were unloaded. Then the trucker retrieving these containers would simply start with the last one stored. Finally, with this solution we have a storage and retrieval process that involves no searching at all and no extra handling of containers.

KEY CONCEPT
A stack can be used to reverse the order of a set of data.

What if the containers were loaded onto the ship in the reverse order they are needed? In that case, unloading the ship will bring the containers off in the order they will be needed and our storage and retrieval process must preserve that order. This would be accomplished by taking the containers to the array for their destination, storing them in the order they were unloaded, and then having the truckers start with the first container stored rather than the last. Again, this solution does not involve any searching or extra handling of containers. This behavior is that of a *queue* data structure, which will be discussed in detail in Chapter 5.

KEY CONCEPT
A queue preserves the order of its data.

Both of our previous solutions, while very efficient, have been dependent upon the order that the containers were placed aboard ship. What do we do if the containers were not placed aboard in any particular order? In that case, we will need to order the destination array by the priority order of the containers. Rather than trying to exhaust this example, let's just say that there are several data structures that might accomplish this purpose including *ordered lists* (Chapter 6), *priority queues* and *heaps* (Chapter 11), and *hash tables* (Chapter 13). We also have the additional issue of how best to organize the file that relates container IDs to the other information about each container. Solutions to this problem might include *binary search trees* (Chapter 10) and *multi-way search trees* (Chapter 12).

Containers as Objects

Our shipping container example illustrates another issue surrounding data structures beyond our discussion of how to store the containers. It also illustrates the issue of what the containers store. Early in our discussion, we made the assumption that all shipping containers have the same shape and same size, and can hold the same volume and type of material. While the first two assumptions must remain true in order for containers to be stored and shipped interchangeably, the latter assumptions may not be true.

For example, consider the shipment of materials that must remain refrigerated. It may be useful to develop refrigerated shipping containers for this purpose. Similarly some products require very strict humidity control and might require a container with additional environmental controls. Others might be designed for hazardous material and may be lined and/or double walled for protection. Once we develop multiple types of containers, then our containers are no longer generic. We can then think of shipping containers as a hierarchy much like a class hierarchy in object-oriented programming. Thus assigning material to a shipping container becomes analogous to assigning an object to a reference. With both shipping containers and objects, not all assignments are valid. We will discuss issues of type compatibility, inheritance, polymorphism, and creating generic data structures in Chapter 3.

Summary of Key Concepts

- Reliable software seldom fails and, when it does, it minimizes the effects of that failure.
- Software systems must be carefully designed, written, and documented to support the work of developers, maintainers, and users.
- Software must make efficient use of resources such as CPU time and memory.
- Quality characteristics must be prioritized, and then maximized to the extent possible.
- A stack can be used to reverse the order of a set of data.
- A queue preserves the order of its data.

Self-Review Questions

SR 1.1 What is the difference between software engineering and programming?

SR 1.2 Name several software quality characteristics.

SR 1.3 What is the relationship between reliability and robustness?

SR 1.4 Describe the benefits of developing software that is maintainable.

SR 1.5 How does the Java programming language address the quality characteristic of portability?

SR 1.6 What is the principle difference in behavior between a stack and a queue?

SR 1.7 List two common data structures that might be used to order a set of data.

Exercises

EX 1.1 Compare and contrast software engineering with other engineering disciplines.

EX 1.2 Give a specific example that illustrates each of the software quality characteristics listed in Figure 1.1.

EX 1.3 Provide an example, and describe the implications, of a trade-off that might be necessary between quality characteristics in the development of a software system.

Answers to Self-Review Questions

SRA 1.1 Software engineering is concerned with the larger goals of system design and development, not just the writing of code. Programmers mature into software engineers as they begin to understand the issues related to the development of high-quality software and adopt the appropriate practices.

SRA 1.2 Software quality characteristics include: correctness, reliability, robustness, usability, maintainability, reusability, portability, and efficiency.

SRA 1.3 Reliability is concerned with how often and under what circumstances a software system fails, and robustness is concerned with what happens when a software system fails.

SRA 1.4 Software that is well structured, well designed, and well documented is much easier to maintain. These same characteristics also provide benefit for distributed development.

SRA 1.5 The Java programming language addresses this issue by compiling into bytecode, which is a low-level language that is not the machine language for any particular CPU. Bytecode runs on a Java Virtual Machine (JVM), which is software that interprets the bytecode and executes it. Therefore, at least theoretically, any system that has a JVM can execute any Java program.

SRA 1.6 A stack reverses order whereas a queue preserves order.

SRA 1.7 Common data structures that can be used to order a set of data include ordered lists, heaps, and hash tables.

2 Analysis of Algorithms

It is important that we understand the concepts surrounding the efficiency of algorithms before we begin building data structures. A data structure built correctly and with an eye toward efficient use of both the CPU and memory is one that can be reused effectively in many different applications. However, a data structure that is not built efficiently is similar to using a damaged original as the master from which to make copies.

CHAPTER OBJECTIVES

- Discuss the goals of software development with respect to efficiency
- Introduce the concept of algorithm analysis
- Explore the concept of asymptotic complexity
- Compare various growth functions

2.1 Algorithm Efficiency

One of the quality characteristics discussed in section 1.1 is the efficient use of resources. One of the most important resources is CPU time. The efficiency of an algorithm we use to accomplish a particular task is a major factor that determines how fast a program executes. Although the techniques that we will discuss here may also be used to analyze an algorithm relative to the amount of memory it uses, we will focus our discussion on the efficient use of processing time.

The *analysis of algorithms* is a fundamental computer science topic and involves a variety of techniques and concepts. It is a primary theme that we return to throughout this book. This chapter introduces the issues related to algorithm analysis and lays the groundwork for using analysis techniques.

KEY CONCEPT

Algorithm analysis is a fundamental computer science topic.

Let's start with an everyday example: washing dishes by hand. If we assume that washing a dish takes 30 seconds and drying a dish takes an additional 30 seconds, then we can see quite easily that it would take n minutes to wash and dry n dishes. This computation could be expressed as follows:

$$\begin{aligned}\text{Time (n dishes)} &= \text{n} * \text{(30 seconds wash time + 30 seconds dry time)}\\ &= \text{60n seconds}\end{aligned}$$

or, written more formally:

$$f(x) = 30x + 30x$$
$$f(x) = 60x$$

On the other hand, suppose we were careless while washing the dishes and splashed too much water around. Suppose each time we washed a dish, we had to dry not only that dish but also all of the dishes we had washed before that one. It would still take 30 seconds to wash each dish, but now it would take 30 seconds to dry the last dish (once), 2 * 30 or 60 seconds to dry the second-to-last dish (twice), 3 * 30 or 90 seconds to dry the third-to-last dish (three times), and so on. This computation could be expressed as follows:

$$\text{Time (n dishes)} = \text{n} * \text{(30 seconds wash time)} + \sum_{i=1}^{n} (i * 30)$$

Using the formula for an arithmetic series $\sum_1^n i = n(n + 1)/2$ then the function becomes

$$\begin{aligned}\text{Time (n dishes)} &= 30n + 30n(n + 1)/2\\ &= 15n^2 + 45n \text{ seconds}\end{aligned}$$

If there were 30 dishes to wash, the first approach would take 30 minutes, whereas the second (careless) approach would take 247.5 minutes. The more dishes we wash the worse that discrepancy becomes. For example, if there were 300 dishes to wash, the first approach would take 300 minutes or 5 hours, whereas the second approach would take 908,315 minutes or roughly 15,000 hours!

2.2 Growth Functions and Big-OH Notation

For every algorithm we want to analyze, we need to define the size of the problem. For our dishwashing example, the size of the problem is the number of dishes to be washed and dried. We also must determine the value that represents efficient use of time or space. For time considerations, we often pick an appropriate processing step that we'd like to minimize, such as our goal to minimize the number of times a dish has to be washed and dried. The overall amount of time spent on the task is directly related to how many times we have to perform that task. The algorithm's efficiency can be defined in terms of the problem size and the processing step.

Consider an algorithm that sorts a list of numbers into increasing order. One natural way to express the size of the problem would be the number of values to be sorted. The processing step we are trying to optimize could be expressed as the number of comparisons we have to make for the algorithm to put the values in order. The more comparisons we make, the more CPU time is used.

KEY CONCEPT

A growth function shows time or space utilization relative to the problem size.

A *growth function* shows the relationship between the size of the problem (n) and the value we hope to optimize. This function represents the *time complexity* or *space complexity* of the algorithm.

The growth function for our second dishwashing algorithm is

$$t(n) = 15n^2 + 45n$$

However, it is not typically necessary to know the exact growth function for an algorithm. Instead, we are mainly interested in the *asymptotic complexity* of an algorithm. That is, we want to focus on the general nature of the function as n increases. This characteristic is based on the *dominant term* of the expression—the term that increases most quickly as n increases. As n gets very large, the value of the dishwashing growth function is dominated by the n^2 term because the n^2 term grows much faster than the n term. The constants, in this case 15 and 45, and the secondary term, in this case 45n, quickly become irrelevant as n increases. That is to say, the value of n^2 dominates the growth in the value of the expression.

The table in Figure 2.1 shows how the two terms and the value of the expression grow. As you can see from the table, as n gets larger, the $15n^2$ term dominates the value of the expression. It is important to note that the 45n term is larger for very small values of n. Saying that a term is the dominant term as n gets large does not mean that it is larger than the other terms for all values of n.

Number of dishes (n)	$15n^2$	45n	$15n^2 + 45n$
1	15	45	60
2	60	90	150
5	375	225	600
10	1,500	450	1,950
100	150,000	4,500	154,500
1,000	15,000,000	45,000	15,045,000
10,000	1,500,000,000	450,000	1,500,450,000
100,000	150,000,000,000	4,500,000	150,004,500,000
1,000,000	15,000,000,000,000	45,000,000	15,000,045,000,000
10,000,000	1,500,000,000,000,000	450,000,000	1,500,000,450,000,000

FIGURE 2.1 Comparison of terms in growth function

The asymptotic complexity is called the *order* of the algorithm. Thus, our second dishwashing algorithm is said to have order n^2 time complexity, written $O(n^2)$. Our first, more efficient dishwashing example, with growth function $t(n) = 60(n)$ would have order n, written O(n). Thus the reason for the difference between our O(n) original algorithm and our $O(n^2)$ sloppy algorithm is the fact each dish will have to be dried multiple times.

This notation is referred to as O() or Big-Oh notation. A growth function that executes in constant time regardless of the size of the problem is said to have O(1). In general, we are only concerned with executable statements in a program or algorithm in determining its growth function and efficiency. Keep in mind, however, that some declarations may include initializations and some of these may be complex enough to factor into the efficiency of an algorithm.

KEY CONCEPT
The order of an algorithm is found by eliminating constants and all but the dominant term in the algorithm's growth function.

As an example, assignment statements and if statements that are only executed once regardless of the size of the problem are O(1). Therefore, it does not matter how many of those you string together; it is still O(1). Loops and method calls may result in higher order growth functions because they may result in a statement or series of statements being executed more than once based on the size of the problem. We will discuss these separately in later sections of this chapter. Figure 2.2 shows several growth functions and their asymptotic complexity.

KEY CONCEPT
The order of an algorithm provides an upper bound to the algorithm's growth function.

More formally, saying that the growth function $t(n) = 15n^2 + 45n$ is $O(n^2)$ means that there exists a constant m and some value of n (n_0), such that $t(n) \leq m^* n^2$ for all $n > n_0$. Another way of stating this is that the order of an algorithm provides an upper bound to its growth function. It is also important to note that there are other related notations such as omega (Ω) which refers to a function that provides a lower

Growth Function	Order	Label
$t(n) = 17$	$O(1)$	constant
$t(n) = 3\log n$	$O(\log n)$	logarithmic
$t(n) = 20n - 4$	$O(n)$	linear
$t(n) = 12n \log n + 100n$	$O(n \log n)$	n log n
$t(n) = 3n^2 + 5n - 2$	$O(n^2)$	quadratic
$t(n) = 8n^3 + 3n^2$	$O(n^3)$	cubic
$t(n) = 2^n + 18n^2 + 3n$	$O(2^n)$	exponential

FIGURE 2.2 Some growth functions and their asymptotic complexity

bound and theta (Θ) which refers to a function that provides both an upper and lower bound. We will focus our discussion on order.

Because the order of the function is the key factor, the other terms and constants are often not even mentioned. All algorithms within a given order are considered to be generally equivalent in terms of efficiency. For example, while two algorithms to accomplish the same task may have different growth functions, if they are both $O(n^2)$ then they are considered to be roughly equivalent with respect to efficiency.

2.3 Comparing Growth Functions

One might assume that, with the advances in the speed of processors and the availability of large amounts of inexpensive memory, algorithm analysis would no longer be necessary. However, nothing could be farther from the truth. Processor speed and memory cannot make up for the differences in efficiency of algorithms. Keep in mind that in our previous discussion we have been eliminating constants as irrelevant when discussing the order of an algorithm. Increasing processor speed simply adds a constant to the growth function. When possible, finding a more efficient algorithm is a better solution than finding a faster processor.

Another way of looking at the effect of algorithm complexity was proposed by Aho, Hopcroft, and Ullman (1974). If a system can currently handle a problem of size n in a given time period, what happens to the allowable size of the problem if we increase the speed of the processor tenfold? As shown in Figure 2.3, the linear case is relatively simple. Algorithm A, with a linear time complexity of n, is indeed improved by a factor of 10, meaning that this algorithm can process 10 times the data in the same amount of time given a tenfold speed up of the processor. However, algorithm B, with a time complexity of n^2, is only improved by a factor of 3.16. Why do we not get the full tenfold increase in problem size? Because the complexity of algorithm B is n^2 our effective speedup is only the square root of 10 or 3.16.

Algorithm	Time Complexity	Max Problem Size Before Speedup	Max Problem Size After Speedup
A	n	s_1	$10s_1$
B	n^2	s_2	$3.16s_2$
C	n^3	s_3	$2.15s_3$
D	2^n	s_4	$s_4 + 3.3$

FIGURE 2.3 Increase in problem size with a tenfold increase in processor speed

KEY CONCEPT
If the algorithm is inefficient, a faster processor will not help in the long run.

Similarly, algorithm C, with complexity n^3, is only improved by a factor of 2.15 or the cube root of 10. For algorithms with *exponential complexity* like algorithm D, in which the size variable is in the exponent of the complexity term, the situation is far worse. In this case the speed up is $\log_2 n$ or in this case, 3.3. Note this is not a factor of 3, but the original problem size plus 3. In the grand scheme of things, if an algorithm is inefficient, speeding up the processor will not help.

Figure 2.4 illustrates various growth functions graphically for relatively small values of n. Note that when n is small, there is little difference between the algo-

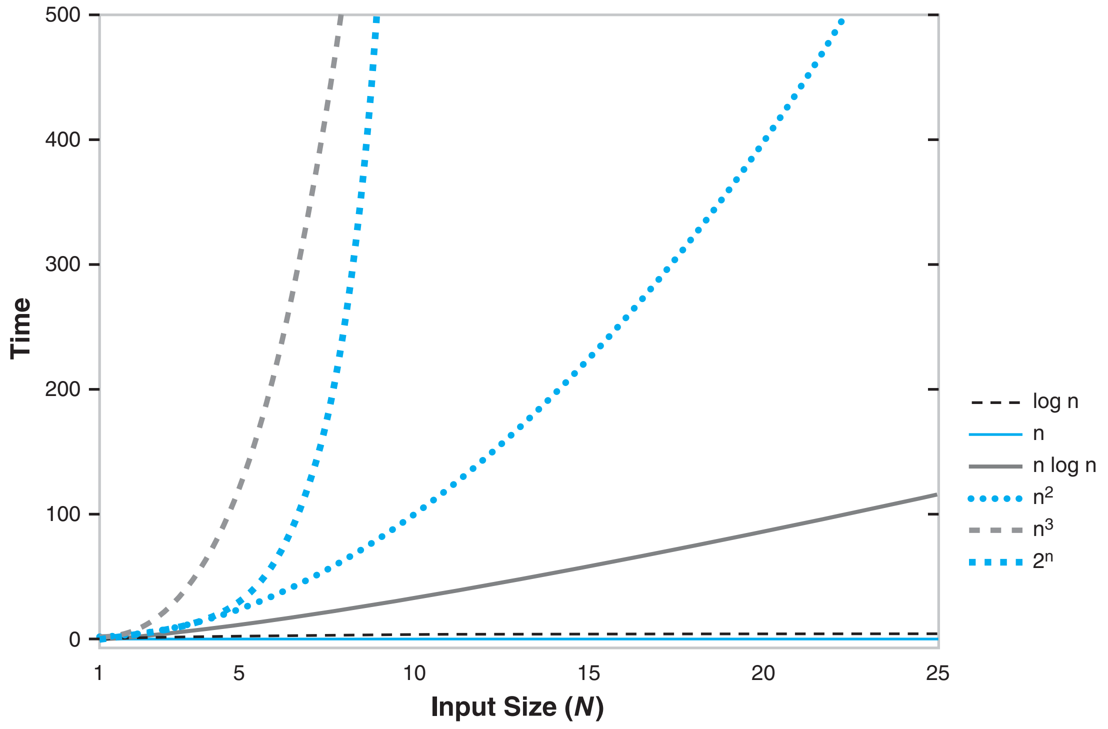


FIGURE 2.4 Comparison of typical growth functions for small values of n

rithms. That is, if you can guarantee a very small problem size (5 or less), it doesn't really matter which algorithm is used. However, notice that in Figure 2.5, as n gets very large, the differences between the growth functions become obvious.

2.4 Determining Time Complexity

Analyzing Loop Execution

To determine the order of an algorithm, we have to determine how often a particular statement or set of statements gets executed. Therefore, we often have to determine how many times the body of a loop is executed. To analyze loop execution, first determine the order of the body of the loop, and then multiply that by the number of times the loop will execute relative to n. Keep in mind that n represents the problem size.

KEY CONCEPT
Analyzing algorithm complexity often requires analyzing the execution of loops.

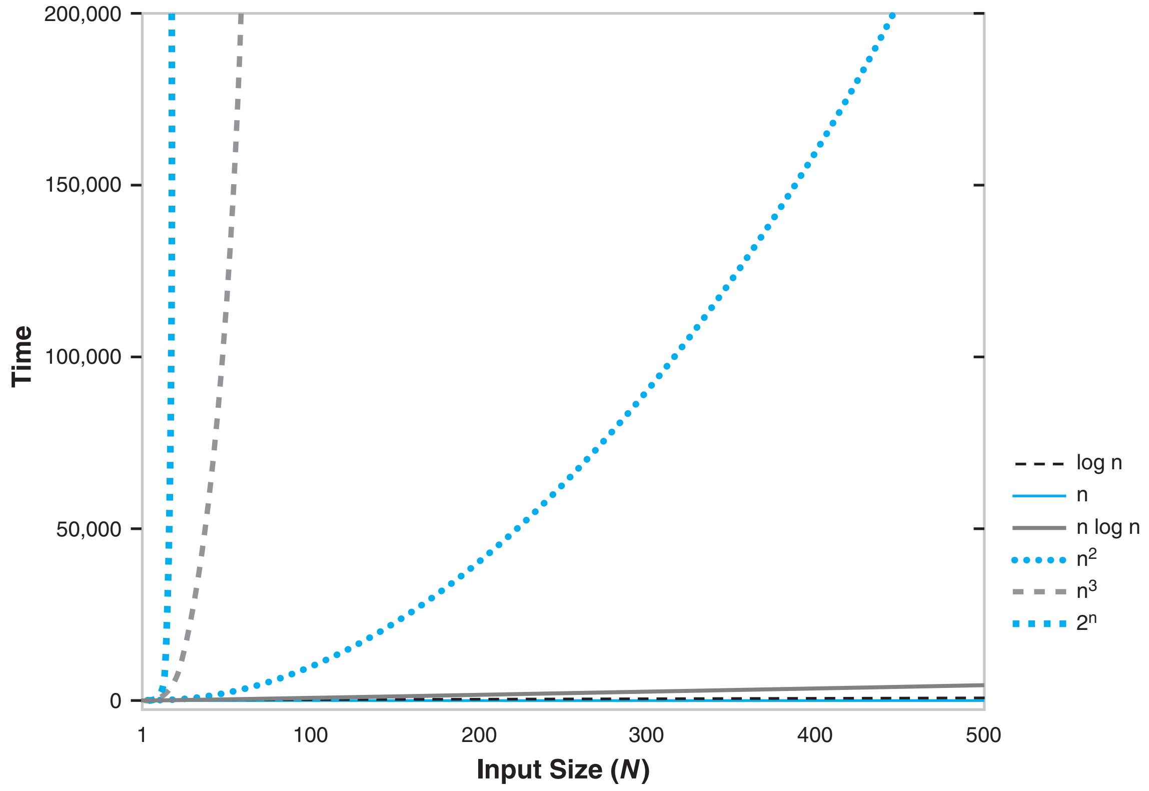


FIGURE 2.5 Comparison of typical growth functions for large values of n

Assuming that the body of a loop is O(1), then a loop such as this:

```
for (int count = 0; count < n; count++)
{
   /* some sequence of O(1) steps */
}
```

would have O(n) time complexity. This is due to the fact that the body of the loop has O(1) complexity but is executed n times by the loop structure. In general, if a loop structure steps through n items in a linear fashion and the body of the loop is O(1), then the loop is O(n). Even in a case where the loop is designed to skip some number of elements, as long as the progression of elements to skip is linear, the loop is still O(n). For example, if the preceding loop skipped every other number (e.g. count += 2), the growth function of the loop would be n/2, but since constants don't affect the asymptotic complexity, the order is still O(n).

Let's look at another example. If the progression of the loop is logarithmic such as the following:

```
count = 1
while (count < n)
{
   count *= 2;
   /* some sequence of O(1) steps */
}
```

then the loop is said to be O(log n). Note that when we use a logarithm in an algorithm complexity, we almost always mean log base 2. This can be explicitly written as O($\log_2 n$). Since each time through the loop the value of `count` is multiplied by 2, the number of times the loop is executed is $\log_2 n$.

KEY CONCEPT
The time complexity of a loop is found by multiplying the complexity of the body of the loop by how many times the loop will execute.

Nested Loops

A slightly more interesting scenario arises when loops are nested. In this case, we must multiply the complexity of the outer loop by the complexity of the inner loop to find the resulting complexity. For example, the following nested loops:

```
for (int count = 0; count < n; count++)
{
   for (int count2 = 0; count2 < n; count2++)
   {
      /* some sequence of O(1) steps */
   }
}
```

would have complexity O(n^2). The body of the inner loop is O(1) and the inner loop will execute n times. This means the inner loop is O(n). Multiplying this result by the number of times the outer loop will execute (n) results in O(n^2).

KEY CONCEPT
The analysis of nested loops must take into account both the inner and outer loops.

What is the complexity of the following nested loop?

```
for (int count = 0; count < n; count++)
{
   for (int count2 = count; count2 < n; count2++)
   {
      /* some sequence of O(1) steps */
   }
}
```

In this case, the inner loop index is initialized to the current value of the index for the outer loop. The outer loop executes n times. The inner loop executes n times the first time, n–1 times the second time, etc. However, remember that we are only interested in the dominant term, not in constants or any lesser terms. If the progression is linear, regardless of whether some elements are skipped, the order is still O(n). Thus the resulting complexity for this code is O(n^2).

Method Calls

Let's suppose that we have the following segment of code:

```
for (int count = 0; count < n; count++)
{
   printsum (count);
}
```

We know from our previous discussion that we find the order of the loop by multiplying the order of the body of the loop by the number of times the loop will execute. In this case, however, the body of the loop is a method call. Therefore, we must first determine the order of the method before we can determine the order of the code segment. Let's suppose that the purpose of the method is to print the sum of the integers from 1 to n each time it is called. We might be tempted to create a brute force method such as the following:

```
public void printsum(int count)
{
   int sum = 0;
   for (int I = 1; I < count; I++)
      sum += I;
   System.out.println (sum);
}
```

What is the time complexity of this printsum method? Keep in mind that only executable statements contribute to the time complexity so in this case, all of the executable statements are O(1) except for the loop. The loop on the other hand is O(n) and thus the method itself is O(n). Now to compute the time complexity of the original loop that called the method, we simply multiply the complexity of the method, which is the body of the loop, by the number of times the loop will execute. Our result, then, is O(n^2) using this implementation of the printsum method.

However, if you recall, we know from our earlier discussion that we do not have to use a loop to calculate the sum of the numbers from 1 to n. In fact, we know that the $\sum_1^n i = n(n + 1)/2$. Now let's rewrite our printsum method and see what happens to our time complexity:

```
public void printsum(int count)
{
   sum = count*(count+1)/2;
   System.out.println (sum);
}
```

Now the time complexity of the printsum method is made up of an assignment statement which is O(1) and a print statement which is also O(1). The result of this change is that the time complexity of the printsum method is now O(1) meaning that the loop that calls this method now goes from being O(n^2) to O(n). We know from our our earlier discussion and from Figure 2.5 that this is a very significant improvement. Once again we see that there is a difference between delivering correct results and doing so efficiently.

What if the body of a method is made up of multiple method calls and loops? Consider the following code using our printsum method above:

```
public void sample(int n)
{
   printsum(n);                                  /* this method call is O(1) */
   for (int count = 0; count < n; count++)       /* this loop is O(n) */
      printsum (count);
   for (int count = 0; count < n; count++)       /* this loop is O(n^2) */
      for (int count2 = 0; count2 < n; count2++)
         System.out.println (count, count2);
}
```

The initial call to the printsum method with the parameter temp is O(1) since the method is O(1). The for loop containing the call to the printsum method with the parameter count is O(n) since the method is O(1) and the loop executes n times. The nested loops are $O(n^2)$ since the inner loop will execute n times each time the outer loop executes and the outer loop will also execute n times. The entire method is then $O(n^2)$ since only the dominant term matters.

More formally, the growth function for the method sample is given by:

$f(x) = 1 + n + n^2$

Then given that we eliminate constants and all but the dominant term, the time complexity is $O(n^2)$.

There is one additional issue to deal with when analyzing the time complexity of method calls and that is recursion, the situation when a method calls itself. We will save that discussion for Chapter 7.

Summary of Key Concepts

- Software must make efficient use of resources such as CPU time and memory.
- Algorithm analysis is a fundamental computer science topic.
- A growth function shows time or space utilization relative to the problem size.
- The order of an algorithm is found by eliminating constants and all but the dominant term in the algorithm's growth function.
- The order of an algorithm provides an upper bound to the algorithm's growth function.
- If the algorithm is inefficient, a faster processor will not help in the long run.
- Analyzing algorithm complexity often requires analyzing the execution of loops.
- The time complexity of a loop is found by multiplying the complexity of the body of the loop by how many times the loop will execute.
- The analysis of nested loops must take into account both the inner and outer loops.

Self-Review Questions

SR 2.1 What is the difference between the growth function of an algorithm and the order of that algorithm?

SR 2.2 Why does speeding up the CPU not necessarily speed up the process by the same amount?

SR 2.3 How do we use the growth function of an algorithm to determine its order?

SR 2.4 How do we determine the time complexity of a loop?

SR 2.5 How do we determine the time complexity of a method call?

Exercises

EX 2.1 What is the order of the following growth functions?

a. $10n^2 + 100n + 1000$

b. $10n^3 - 7$

c. $2^n + 100n^3$

d. $n^2 \log n$

EX 2.2 Arrange the growth functions of the previous exercise in ascending order of efficiency for n=10 and again for n=1,000,000.

EX 2.3 Write the code necessary to find the largest element in an unsorted array of integers. What is the time complexity of this algorithm?

EX 2.4 Determine the growth function and order of the following code fragment:

```
for (int count=0; count < n; count++)
{
   for (int count2=0; count2 < n; count2=count2+2)
   {
      System.out.println(count, count2);
   }
}
```

EX 2.5 Determine the growth function and order of the following code fragment:

```
for (int count=0; count < n; count++)
{
   for (int count2=0; count2 < n; count2=count2*2)
   {
      System.out.println(count, count2);
   }
}
```

EX 2.6 The table in Figure 2.1 shows how the terms of the growth function for our dishwashing example relate to one another as n grows. Write a program that will create such a table for any given growth function.

Answers to Self-Review Questions

SRA 2.1 The growth function of an algorithm represents the exact relationship between the problem size and the time complexity of the solution. The order of the algorithm is the asymptotic time complexity. As the size of the problem grows, the complexity of the algorithm approaches the asymptotic complexity.

SRA 2.2 Linear speedup only occurs if the algorithm has constant order, O(1), or linear order, O(n). As the complexity of the algorithm grows, faster processors have significantly less impact.

SRA 2.3 The order of an algorithm is found by eliminating constants and all but the dominant term from the algorithm's growth function.

SRA 2.4 The time complexity of a loop is found by multiplying the time complexity of the body of the loop times the number of times the loop will execute.

SRA 2.5 The time complexity of a method call is found by determining the time complexity of the method and then substituting that for the method call.

References

Aho, A. V., J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Reading, Mass.: Addison-Wesley, 1974.

Collections

3

This chapter begins our exploration of collections and the underlying data structures used to implement them. It lays the groundwork for the study of collections by carefully defining the issues and goals related to their design. This chapter also introduces a collection called a stack and uses it to exemplify the issues related to the design, implementation, and use of collections.

CHAPTER OBJECTIVES

- Define the concepts and terminology related to collections
- Explore the basic structure of the Java Collections API
- Discuss the abstract design of collections
- Explore issues surrounding collections including inheritance, polymorphism, generics, and interfaces
- Define a stack collection
- Use a stack collection to solve a problem
- Examine an array implementation of a stack

3.1 Introduction to Collections

KEY CONCEPT
A collection is an object that gathers and organizes other objects.

A *collection* is an object that gathers and organizes other objects. It defines the specific ways in which those objects, which are called *elements* of the collection, can be accessed and managed. The user of a collection, which is usually another class or object in the software system, must interact with the collection only in the prescribed ways.

Over the past 50 years, several specific types of collections have been defined by software developers and researchers. Each type of collection lends itself to solving particular kinds of problems. A large portion of this book is devoted to exploring these classic collections.

KEY CONCEPT
Elements in a collection are typically organized by the order of their addition to the collection or by some inherent relationship among the elements.

Collections can be separated into two broad categories: linear and nonlinear. As the name implies, a *linear collection* is one in which the elements of the collection are organized in a straight line. A *nonlinear collection* is one in which the elements are organized in something other than a straight line, such as a hierarchy or a network. For that matter, a nonlinear collection may not have any organization at all.

Figure 3.1 shows a linear and a nonlinear collection. It usually doesn't matter whether the elements in a linear collection are depicted horizontally or vertically.

The organization of the elements in a collection, relative to each other, is usually determined by one of two things:

- The order in which they were added to the collection
- Some inherent relationship among the elements themselves

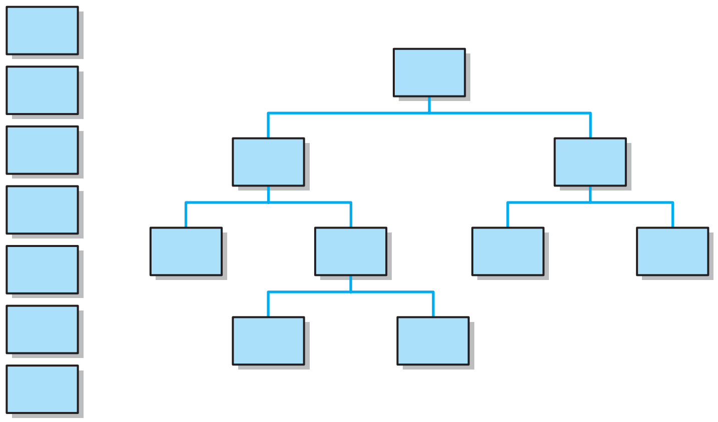


FIGURE 3.1 A linear and a nonlinear collection

For example, one linear collection may always add new elements to one end of the line, so the order of the elements is determined by the order in which they are added. Another linear collection may be kept in sorted order based on some characteristic of the elements. For example, a list of people may be kept in alphabetical order based on the characters that make up their name. The specific organization of the elements in a nonlinear collection can be determined in either of these two ways as well.

Abstract Data Types

An *abstraction* hides certain details at certain times. Dealing with an abstraction is easier than dealing with too many details at one time. In fact, we couldn't get through a day without relying on abstractions. For example, we couldn't possibly drive a car if we had to worry about all the details that make the car work: the spark plugs, the pistons, the transmission, and so on. Instead, we can focus on the *interface* to the car: the steering wheel, the pedals, and a few other controls. These controls are an abstraction, hiding the underlying details and allowing us to control an otherwise very complicated machine.

A collection, like any well-designed object, is an abstraction. A collection defines the interface operations through which the user can manage the objects in the collection, such as adding and removing elements. The user interacts with the collection through this interface, as depicted in Figure 3.2. However, the details of how a collection is implemented to fulfill that definition are another issue altogether. A class that implements the collection's interface must fulfill the conceptual definition of the collection, but it can do so in many ways.

KEY CONCEPT
A collection is an abstraction where the details of the implementation are hidden.

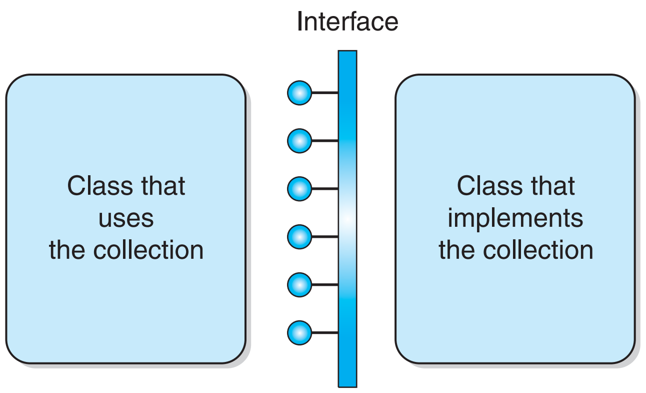

Interface
Class that uses the collection
Class that implements the collection

FIGURE 3.2 A well-defined interface masks the implementation of the collection

Abstraction is another important software engineering concept. In large software systems, it is virtually impossible for any one person to grasp all of the details of the system at once. Instead, the system is divided into abstract subsystems such that the purpose of and the interactions among those subsystems can be specified. Subsystems may then be assigned to different developers or groups of developers that will develop the subsystem to meet its specification.

An object is the perfect mechanism for creating a collection because, if it is designed correctly, the internal workings of an object are *encapsulated* from the rest of the system. In almost all cases, the instance variables defined in a class should be declared with private visibility. Therefore, only the methods of that class can access and modify them. The only interaction a user has with an object should be through its public methods, which represent the services that the object provides.

As we progress through our exploration of collections, we will always stress the idea of separating the interface from the implementation. Therefore, for every collection that we examine, we should consider the following:

- How does the collection operate, conceptually?
- How do we formally define the interface to the collection?
- What kinds of problems does the collection help us solve?
- In which various ways might we implement the collection?
- What are the benefits and costs of each implementation?

Before we continue, let's carefully define some other terms related to the exploration of collections. A *data type* is a group of values and the operations defined on those values. The primitive data types defined in Java are the primary examples. For example, the integer data type defines a set of numeric values and the operations (addition, subtraction, etc.) that can be used on them.

An *abstract data type* (ADT) is a data type whose values and operations are not inherently defined within a programming language. It is abstract only in that the details of its implementation must be defined and should be hidden from the user. A collection, therefore, is an abstract data type.

A *data structure* is the collection of programming constructs used to implement a collection. For example, a collection might be implemented using a fixed-size structure such as an array. One interesting artifact of these definitions and our design decision to separate the interface from the implementation (i.e., the collection from the data structure that implements it) is that we may, and often do, end up with a linear data structure, such as an array, being used to implement a nonlinear collection, such as a tree.

KEY CONCEPT

A data structure is the underlying programming construct used to implement a collection.

Historically, the terms ADT and data structure have been used in various ways. We carefully define them here to avoid any confusion, and will use them consistently. Throughout this book we will examine various data structures and how they can be used to implement various collections.

The Java Collections API

The Java programming language is accompanied by a very large library of classes that can be used to support the development of software. Parts of the library are organized into *application programming interfaces* (APIs). The *Java Collections API* is a set of classes that represent a few specific types of collections, implemented in various ways.

You might ask why we should learn how to design and implement collections if a set of collections has already been provided for us. There are several reasons. First, the Java Collections API provides only a subset of the collections you may want to use. Second, the classes that are provided may not implement the collections in the ways you desire. Third, and perhaps most important, the study of software development requires a deep understanding of the issues involved in the design of collections and the data structures used to implement them.

As we explore various types of collections, we will also examine the appropriate classes of the Java Collections API. In each case, we will analyze the various implementations that we develop and compare them to the approach used by the classes in the standard library.

3.2 A Stack Collection

Let's look at an example of a collection. A *stack* is a linear collection whose elements are added and removed from the same end. We say that a stack is processed in a *last in, first out* (LIFO) manner. That is, the last element to be put on a stack will be the first one that gets removed. Said another way, the elements of a stack are removed in the reverse order of their placement on it. In fact, one of the principal uses of a stack in computing is to reverse the order of something (e.g., an undo operation).

KEY CONCEPT

Stack elements are processed in a LIFO manner—the last element in is the first element out.

The processing of a stack is shown in Figure 3.3. Usually a stack is depicted vertically, and we refer to the end to which elements are added and from which they are removed as the *top* of the stack.

Recall from our earlier discussions that we define an abstract data type (ADT) by identifying a specific set of operations that establishes the valid ways in which we can manage the elements stored in the data structure. We always want to use this concept to formally define the operations for a collection and work within

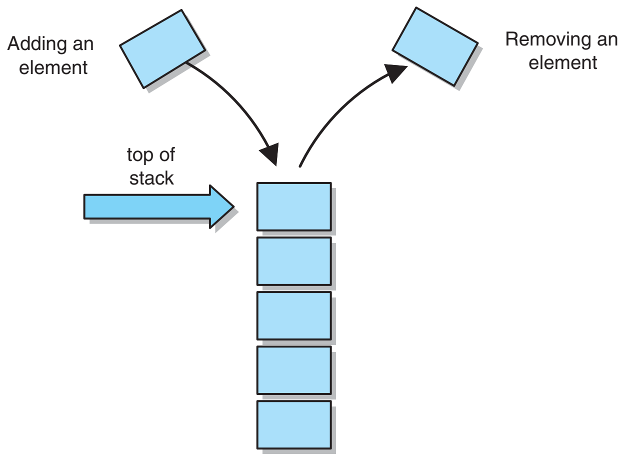


FIGURE 3.3 A conceptual view of a stack

the functionality it provides. That way, we can cleanly separate the interface to the collection from any particular implementation technique used to create it.

The operations for a stack ADT are listed in Figure 3.4. In stack terminology, we *push* an element onto a stack and we *pop* an element off a stack. We can also *peek* at the top element of a stack, examining it or using it as needed, without actually removing it from the collection. There are also the general operations that allow us to determine if the stack is empty and, if not empty, how many elements it contains.

KEY CONCEPT
A programmer should choose the structure that is appropriate for the type of data management needed.

Sometimes there are variations on the naming conventions for the operations on a collection. For a stack, the use of the terms push and pop is relatively standard. The `peek` operation is sometimes referred to as *top*.

Operation	Description
`push`	Adds an element to the top of the stack.
`pop`	Removes an element from the top of the stack.
`peek`	Examines the element at the top of the stack.
`isEmpty`	Determines if the stack is empty.
`size`	Determines the number of elements on the stack.

FIGURE 3.4 The operations on a stack

DESIGN FOCUS

In the design of the stack ADT, we see the separation between the role of the stack and the role of the application that is using the stack. Notice that any implementation of this stack ADT is expected to throw an exception if a `pop` or `peek` operation is requested on an empty stack. The role of the collection is not to determine how such an exception is handled but merely to report it back to the application using the stack. Similarly, the concept of a full stack does not exist in the stack ADT. Thus, it is the role of the stack collection to manage its own storage to eliminate the possibility of being full.

Keep in mind that the definition of a collection is not universal. You will find variations in the operations defined for specific data structures from one book to another. We've been very careful in this book to define the operations on each collection so that they are consistent with its purpose.

For example, note that none of the stack operations in Figure 3.4 allow us to reach down into the stack to modify, remove, or reorganize the elements in the stack. That is the very nature of a stack—all activity occurs at one end. If we discover that, to solve a particular problem, we need to access the elements in the middle or at the bottom of the collection, then a stack is not the appropriate data structure to use.

We do provide a `toString` operation for the collection. This is not a classic operation defined for a stack, and it could be argued that this operation violates the prescribed behavior of a stack. However, it provides a convenient means to traverse and display the stack's contents without allowing modification of the stack and this is quite useful for debugging purposes.

3.3 Crucial OO Concepts

Now let's consider what we will store in our stack. One possibility would be to simply recreate our stack data structure each time we need it and create it to store the specific object type for that application. For example, if we needed a stack of strings, we would simply copy and paste our stack code and change the object type to `String`. While copy, paste, and modify is technically a form of reuse, this brute force type of reuse is not our goal. Reuse, in its purest form, should mean that we create a collection that is written once, compiled into byte code once, and will then handle any objects we choose to store in it safely, efficiently, and effectively. To accomplish these goals, we must take *type compatibility* and *type checking* into account. Type compatibility is the concept of whether a particular assignment of an object to a reference is legal. For example, the following assignment is not legal because you cannot assign a reference declared to be of type `String` to point to an object of type `Integer`.

```
String x = new Integer(10);
```

Java provides compile-time type checking that will flag this invalid assignment. A second possibility of what to store in our collection would be to take advantage of the concepts of *inheritance* and *polymorphism* to create a collection that could store objects of any class. To explore this possibility, we must first explore the concept of inheritance.

Inheritance

KEY CONCEPT
Inheritance is the process of deriving a new class from an existing one.

KEY CONCEPT
One purpose of inheritance is to reuse existing software.

Inheritance is the process in which a new class is derived from an existing one. The new class automatically contains some or all of the variables and methods in the original class. Then, to tailor the class as needed, the programmer can add new variables and methods to the derived class, or modify the inherited ones.

In general, creating new classes via inheritance is faster, easier, and cheaper than writing them from scratch. At the heart of inheritance is the idea of *software reuse*. By using existing software components to create new ones, we capitalize on all of the effort that went into the design, implementation, and testing of the existing software.

Keep in mind that the word *class* comes from the idea of classifying groups of objects with similar characteristics. Classification schemes often use levels of classes that relate to one another. For example, all mammals share certain characteristics: they are warm-blooded, have hair, and bear live offspring. Now consider a subset of mammals, such as horses. All horses are mammals, and have all the characteristics of mammals. But they also have unique features that make them different from other mammals.

If we map this idea into software terms, a class called `Mammal` would have certain variables and methods that describe the state and behavior of mammals. A `Horse` class could be derived from the existing `Mammal` class, automatically inheriting the variables and methods contained in `Mammal`. The `Horse` class can refer to the inherited variables and methods as if they had been declared locally in that class. New variables and methods can then be added to the derived class, to distinguish a horse from other mammals. Inheritance nicely models many situations found in the natural world.

The original class that is used to derive a new one is called the *parent class*, *superclass*, or *base class*. The derived class is called a *child class*, or *subclass*. Java uses the reserved word `extends` to indicate that a new class is being derived from an existing class.

KEY CONCEPT
Inherited variables and methods can be used in the derived class as if they had been declared locally.

The derivation process should establish a specific kind of relationship between two classes: an *is-a relationship*. This type of relationship means that the derived class should be a more specific version of the original. For example, a horse is a mammal. Not all mammals are horses, but all horses are mammals.

KEY CONCEPT
Inheritance creates an is-a relationship between all parent and child classes.

Let's look at an example. The following class can be used to define a book:

```
class Book
{
   protected int numPages;

   protected void pages()
   {
      System.out.println ("Number of pages: " + numPages);
   }
}
```

To derive a child class that is based on the `Book` class, we use the reserved word `extends` in the header of the child class. For example, a `Dictionary` class can be derived from `Book` as follows:

```
class Dictionary extends Book
{
   private int numDefs;

   public void info()
   {
      System.out.println ("Number of definitions: " + numDefs);
      System.out.println ("Definitions per page: "
                                   + numDefs/numPages);
   }
}
```

By saying that the `Dictionary` class extends the `Book` class, the `Dictionary` class automatically inherits the `numPages` variable and the `pages` method. Note that the `info` method uses the `numPages` variable explicitly.

Inheritance is a one-way street. The `Book` class cannot use variables or methods that are declared explicitly in the `Dictionary` class. For instance, if we created an object from the `Book` class, it could not be used to invoke the `info` method. This restriction makes sense, because a child class is a more specific version of the parent. A dictionary has pages, because all books have pages; however, although a dictionary has definitions, not all books do.

Throughout this book, we will use the Unified Modeling Language (UML) to represent the design of our systems. Appendix A provides an introduction to the UML. Inheritance relationships are represented in UML class diagrams using an arrow with an open arrowhead pointing from the child class to the parent class.

Class Hierarchies

A child class derived from one parent can be the parent of its own child class. Furthermore, multiple classes can be derived from a single parent. Therefore, inheritance relationships often develop into *class hierarchies*. The UML class diagram in Figure 3.5 shows a class hierarchy that incorporates the inheritance relationship between classes `Mammal` and `Horse`.

KEY CONCEPT
The child of one class can be the parent of one or more other classes, creating a class hierarchy.

There is no limit to the number of children a class can have, or to the number of levels to which a class hierarchy can extend. Two children of the same parent are called *siblings*. Although siblings share the characteristics passed on by their common parent, they are not related by inheritance, because one is not used to derive the other.

In class hierarchies, common features should be kept as high in the hierarchy as reasonably possible. That way, the only characteristics explicitly established in a child class are those that make the class distinct from its parent and from its siblings. This approach maximizes the ability to reuse classes. It also facilitates maintenance activities, because when changes are made to the parent, they are automatically reflected in the descendants. Always remember to maintain the is-a relationship when building class hierarchies.

KEY CONCEPT
Common features should be located as high in a class hierarchy as is reasonable, minimizing maintenance efforts.

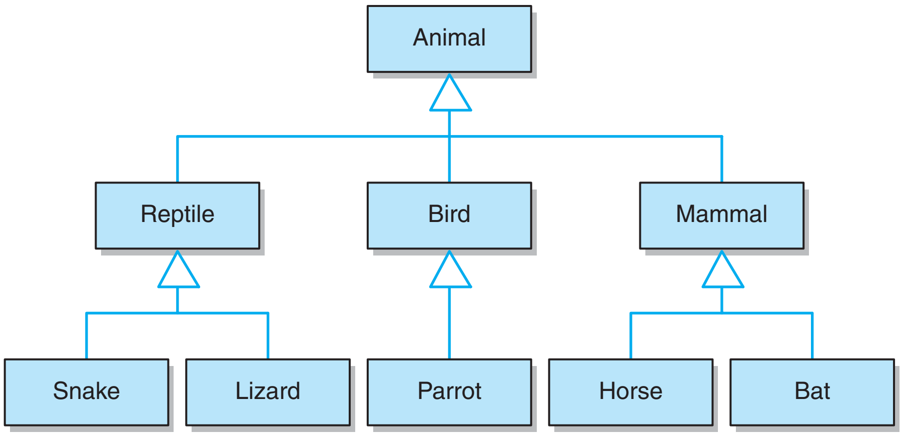


FIGURE 3.5 A UML class diagram showing a class hierarchy

The inheritance mechanism is transitive. That is, a parent passes along a trait to a child class, that child class passes it along to its children, and so on. An inherited feature might have originated in the immediate parent, or possibly from several levels higher in a more distant ancestor class.

There is no single best hierarchy organization for all situations. The decisions made when designing a class hierarchy restrict and guide more detailed design decisions and implementation options, and they must be made carefully.

The `Object` Class

In Java, all classes are derived ultimately from the `Object` class. If a class definition doesn't use the `extends` clause to derive itself explicitly from another class, then that class is automatically derived from the `Object` class by default. Therefore, the following two class definitions are equivalent:

```
class Thing
{
   // whatever
}
```

and

```
class Thing extends Object
{
   // whatever
}
```

Because all classes are derived from `Object`, any public method of `Object` can be invoked through any object created in any Java program. The `Object` class is defined in the `java.lang` package of the standard class library.

The `toString` method, for instance, is defined in the `Object` class, so the `toString` method can be called on any object. When a `println` method is called with an object parameter, `toString` is called to determine what to print.

KEY CONCEPT
All Java classes are derived, directly or indirectly, from the `Object` class.

The definition for `toString` that is provided by the `Object` class returns a string containing the object's class name followed by a numeric value that is unique for that object. Usually, we override the `Object` version of `toString` to fit our own needs. The `String` class has overridden the `toString` method so that it returns its stored string value.

The `equals` method of the `Object` class is also useful. Its purpose is to determine if two objects are equal. The definition of the `equals` method provided by the `Object` class returns true if the two object references actually refer to the same object (that is, if they are aliases). Classes often override the inherited

KEY CONCEPT

The toString and equals methods are defined in the Object class and therefore are inherited by every class in every Java program.

definition of the equals method in favor of a more appropriate definition. For instance, the String class overrides equals so that it returns true only if both strings contain the same characters in the same order. A more complete discussion of inheritance is provided in Appendix B.

Polymorphism

This leads us to the concept of polymorphism. The term *polymorphism* can be defined as "having many forms." A *polymorphic reference* is a reference variable that can refer to different types of objects at different points in time. The specific method invoked through a polymorphic reference can change from one invocation to the next.

Consider the following line of code:

```
obj.doIt();
```

KEY CONCEPT

A polymorphic reference can refer to different types of objects over time.

If the reference obj is polymorphic, it can refer to different types of objects at different times. If that line of code is in a loop or in a method that is called more than once, that line of code might call a different version of the doIt method each time it is invoked.

At some point, the commitment is made to execute certain code to carry out a method invocation. This commitment is referred to as *binding* a method invocation to a method definition. In most situations, the binding of a method invocation to a method definition can occur at compile time. For polymorphic references, however, the decision cannot be made until run time. The method definition that is used is based on the object that is being referred to by the reference variable at that moment. This deferred commitment is called *late binding* or *dynamic binding*. It is less efficient than binding at compile time because the decision has to be made during the execution of the program. This overhead is generally acceptable in light of the flexibility that a polymorphic reference provides.

There are two ways to create a polymorphic reference in Java: using inheritance and using interfaces. The following sections describe creating polymorphism via inheritance. We will revisit the discussion of polymorphism via interfaces in Chapter 6.

References and Class Hierarchies

In Java, a reference that is declared to refer to an object of a particular class also can be used to refer to an object of any class related to it by inheritance. For example, if the class Mammal is used to derive the class Horse, then a Mammal reference can be used to refer to an object of class `Horse`. This ability is shown in the code segment below:

```
Mammal pet;
Horse secretariat = new Horse();
pet = secretariat;  // a valid assignment
```

The reverse operation, assigning the `Mammal` object to a `Horse` reference, is also valid, but requires an explicit cast. Assigning a reference in this direction is generally less useful and more likely to cause problems, because although a horse has all the functionality of a mammal (because a horse *is-a* mammal), the reverse is not necessarily true.

KEY CONCEPT
A reference variable can refer to any object created from any class related to it by inheritance.

This relationship works throughout a class hierarchy. If the `Mammal` class were derived from a class called `Animal`, then the following assignment would also be valid:

```
Animal creature = new Horse();
```

The reference variable `creature`, as defined in the previous section, can be polymorphic, because at any point in time it could refer to an `Animal` object, a `Mammal` object, or a `Horse` object. Suppose that all three of these classes have a method called `move` and that it is implemented in a different way in each class (because the child class overrode the definition it inherited). The following invocation calls the `move` method, but the particular version of the method it calls is determined at run time:

```
creature.move();
```

At the point when this line is executed, if `creature` currently refers to an `Animal` object, the `move` method of the `Animal` class is invoked. Likewise, if `creature` currently refers to a `Mammal` or `Horse` object, the `Mammal` or `Horse` version of `move` is invoked, respectively. Again, a more complete discussion of polymorphism and all of its implications is included in Appendix B.

KEY CONCEPT
A polymorphic reference uses the type of the object, not the type of the reference, to determine which version of a method to invoke.

Carrying this to the extreme, an `Object` reference can be used to refer to any object, because ultimately all classes are descendants of the `Object` class. An `ArrayList`, for example, uses polymorphism in that it is designed to hold `Object` references. That's why an `ArrayList` can be used to store any kind of object. In fact, a particular `ArrayList` can be used to hold several different types of objects at one time, because, in essence, they are all `Object` objects.

The result of this discussion would seem to be that we could simply store `Object` references in our stack and take advantage of polymorphism via inheritance to create a collection that can store any type of objects. However, this possible solution creates some unexpected consequences. Because in this chapter we focus on implementing a stack with an array, let's examine what can happen when dealing with polymorphic references and arrays. Consider our classes represented in Figure 3.5. Since `Animal` is a superclass of all of the other classes in this diagram, an assignment such as the following is allowable:

```
Animal creature = new Bird();
```

However, this also means that the following assignments will compile as well:

```
Animal[] creatures = new Mammal[];
creatures[1] = new Reptile();
```

Note that by definition, `creatures[1]` should be both a `Mammal` and an `Animal`, but not a `Reptile`. This code will compile but will generate a `java.lang.ArrayStoreException` at run time. Thus, because using the `Object` class will not provide us with compile-time type checking, we should look for a better solution.

Generics

Beginning with Java 5.0, Java enables us to define a class based on a *generic type*. That is, we can define a class so that it stores, operates on, and manages objects whose type is not specified until the class is instantiated. Generics are an integral part of our discussions of collections and their underlying implementations throughout the rest of this book.

Let's assume we need to define a class called `Box` that stores and manages other objects. As we discussed, using polymorphism, we could simply define `Box` so that internally it stores references to the `Object` class. Then, any type of object could be stored inside a box. In fact, multiple types of unrelated objects could be stored in `Box`. We lose a lot of control with that level of flexibility in our code.

A better approach is to define the `Box` class to store a generic type `T`. (We can use any identifier we want for the generic type, but using `T` has become a convention.) The header of the class contains a reference to the type in angle brackets. For example:

```
class Box<T>
{
   // declarations and code that manage objects of type T
}
```

Then, when a `Box` is needed, it is instantiated with a specific class used in place of `T`. For example, if we wanted a `Box` of `Widget` objects, we could use the following declaration:

```
Box<Widget> box1 = new Box<Widget>;
```

The type of the `box1` variable is `Box<Widget>`. In essence, for the `box1` object, the `Box` class replaces `T` with `Widget`. Now suppose we wanted a `Box` in which to store `Gadget` objects; we could make the following declaration:

```
Box<Gadget> box2 = new Box<Gadget>;
```

For `box2`, the `Box` class essentially replaces `T` with `Gadget`. So, although the `box1` and `box2` objects are both boxes, they have different types because the generic type is taken into account. This is a safer implementation, because at this point we cannot use `box1` to store gadgets (or anything else for that matter), nor could we use `box2` to store widgets. A generic type such as `T` cannot be instantiated. It is merely a placeholder to allow us to define the class that will manage a specific type of object that is established when the class is instantiated.

Given that we now have a mechanism using generic types for creating a collection that can be used to store any type of object safely and effectively, let's continue on with our discussion of the stack collection.

3.4 A Stack ADT

Interfaces

To facilitate the separation of the interface operations from the methods that implement them, we can define a Java interface structure for a collection. A Java interface provides a formal mechanism for defining the set of operations for any collection.

KEY CONCEPT

A Java interface defines a set of abstract methods and is useful in separating the concept of an abstract data type from its implementation.

Recall that a Java interface defines a set of abstract methods, specifying each method's signature but not its body. A class that implements an interface provides definitions for the methods defined in the interface. The interface name can be used as the type of a reference, which can be assigned any object of any class that implements the interface.

Listing 3.1 defines a Java interface for a stack collection. We name a collection interface using the collection name followed by the abbreviation ADT (for abstract data type). Thus, `StackADT.java` contains the interface for a stack collection. It is defined as part of the `jss2` package, which contains all of the collection classes and interfaces presented in this book.

Note that the stack interface is defined as `StackADT<T>`, operating on a generic type `T`. In the methods of the interface, the type of various parameters and return values is often expressed using the generic type `T`. When this interface is implemented, it will be based on a type that is substituted for `T`.

KEY CONCEPT

By using the interface name as a return type, the interface doesn't commit the method to the use of any particular class that implements a stack.

Later in this chapter we examine a class that implements this interface. In the next chapter, we will examine an alternative implementation. For

LISTING 3.1

```
/**
 *  @author Lewis and Chase
 *
 *  Defines the interface to a stack data structure.
 */

package jss2;

public interface StackADT<T>
{
   /** Adds one element to the top of this stack.
    *  @param element element to be pushed onto stack
    */

   public void push (T element);
   /** Removes and returns the top element from this stack.
    *  @return T element removed from the top of the stack
    */

   public T pop();
   /** Returns without removing the top element of this stack.
    *  @return T element on top of the stack
    */

   public T peek();
   /** Returns true if this stack contains no elements.
    *  @return boolean whether or not this stack is empty
    */

   public boolean isEmpty();
   /** Returns the number of elements in this stack.
    *  @return int number of elements in this stack
    */

   public int size();
   /** Returns a string representation of this stack.
    *  @return String representation of this stack
    */

   public String toString();
}
```

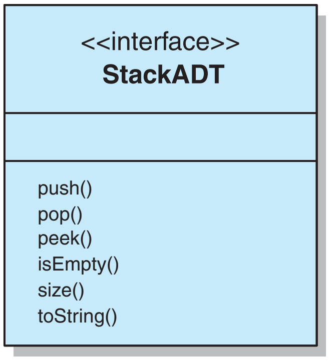


FIGURE 3.6 The `StackADT` interface in UML

now, our abstract understanding of how a stack operates allows us to explore situations in which stacks help us solve particular problems.

Each time we introduce an interface, a class, or a system in this text, we will accompany that description with the UML description of that interface, class, or system. This should help you become accustomed to reading UML descriptions and creating them for other classes and systems. Figure 3.6 illustrates the UML description of the `StackADT` interface. Note that UML provides flexibility in describing the methods associated with a class or interface. In this case, we have chosen to identify each of the methods as public (+), but we have not listed the parameters for each.

Stacks are used quite frequently in the computing world. For example, the undo operation in a word processor is usually implemented using a stack. As we make changes to a document (add data, delete data, make format changes, etc.), the word processor keeps track of each operation by pushing some representation of it onto a stack. If we choose to undo an operation, the word processing software pops the most recently performed operation off the stack and reverses it. If we choose to undo again (undoing the second-to-last operation we performed),

DESIGN FOCUS

Undo operations are often implemented using a special type of stack called a drop-out stack. The basic operations on a drop-out stack are the same as those for a stack (i.e., `push`, `pop`, and `peek`). The only difference is that a drop-out stack has a limit to the number of elements it will hold and, once that limit is reached, the element on the bottom of the stack drops off the stack when a new element is pushed on. The development of a drop-out stack is left as an exercise.

another element is popped from the stack. In most word processors, many operations can be reversed in this manner.

The following section explores in detail an example of using a stack to solve a problem.

3.5 Using Stacks: Evaluating Postfix Expressions

Traditionally, arithmetic expressions are written in *infix* notation, meaning that the operator is placed between its operands in the form

<operand> <operator> <operand>

such as in the expression

```
4 + 5
```

When evaluating an infix expression, we rely on precedence rules to determine the order of operator evaluation. For example, the expression

```
4 + 5 * 2
```

evaluates to 14 rather than 18 because of the precedence rule that says in the absence of parentheses, multiplication evaluates before addition.

In a *postfix* expression, the operator comes after its two operands. Therefore, a postfix expression takes the form

<operand> <operand> <operator>

For example, the postfix expression

```
6 9 -
```

is equivalent to the infix expression

```
6 - 9
```

A postfix expression is generally easier to evaluate than an infix expression because precedence rules and parentheses do not have to be taken into account. The order of the values and operators in the expression are sufficient to determine the result. Programming language compilers and run-time environments often use postfix expressions in their internal calculations for this reason.

The process of evaluating a postfix expression can be stated in one simple rule: Scanning from left to right, apply each operation to the two operands immediately preceding it and replace the operator with the result. At the end we are left with the final value of the expression.

Consider the infix expression we looked at earlier:

```
4 + 5 * 2
```

In postfix notation, this expression would be written

```
4 5 2 * +
```

Let's use our evaluation rule to determine the final value of this expression. We scan from the left until we encounter the multiplication (`*`) operator. We apply this operator to the two operands immediately preceding it (`5` and `2`) and replace it with the result (`10`), leaving us with

```
4 10 +
```

Continuing our scan from left to right, we immediately encounter the plus (`+`) operator. Applying this operator to the two operands immediately preceding it (`4` and `10`) yields `14`, which is the final value of the expression.

Let's look at a slightly more complicated example. Consider the following infix expression:

```
(3 * 4 – (2 + 5)) * 4 / 2
```

The equivalent postfix expression is

```
3 4 * 2 5 + – 4 * 2 /
```

Applying our evaluation rule results in:

```
        12 2 5 + – 4 * 2 /
then    12 7 – 4 * 2 /
then    5 4 * 2 /
then    20 2 /
then    10
```

Now let's think about the design of a program that will evaluate a postfix expression. The evaluation rule relies on being able to retrieve the previous two operands whenever we encounter an operator. Furthermore, a large postfix expression will have many operators and operands to manage. It turns out that a stack is the perfect collection to use in this case. The operations provided by a stack coincide nicely with the process of evaluating a postfix expression.

KEY CONCEPT
A stack is the ideal data structure to use when evaluating a postfix expression.

The algorithm for evaluating a postfix expression using a stack can be expressed as follows: Scan the expression from left to right, identifying each token (operator or operand) in turn. If it is an operand, push it onto the stack. If it is an operator, pop the top two elements off of the stack, apply the operation to them, and push the result onto the stack. When we reach the end of the expression, the element remaining on the stack is the result of the expression. If at any point we attempt to pop two elements off of the stack but there are not two elements on the stack, then our postfix expression was not properly formed. Similarly, if we reach the end of the expression and more than one element remains on the stack, then our expression was not well formed. Figure 3.7 depicts the use of a stack to evaluate a postfix expression.

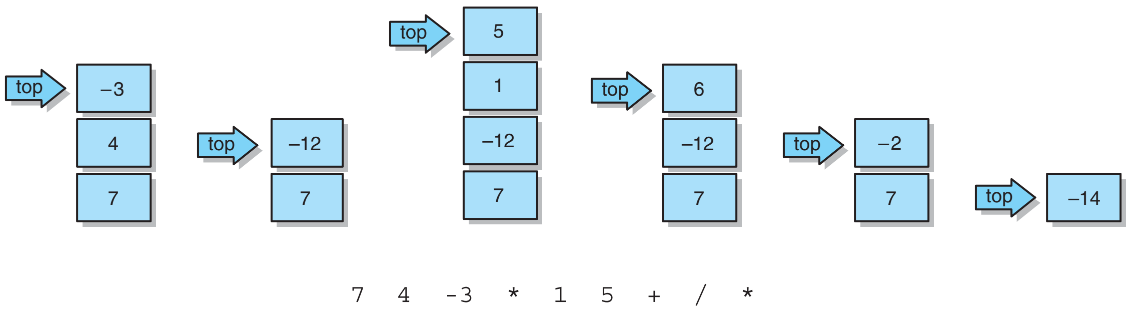


FIGURE 3.7 Using a stack to evaluate a postfix expression

The program in Listing 3.2 evaluates multiple postfix expressions entered by the user. It uses the `PostfixEvaluator` class shown in Listing 3.3.

To keep things simple, this program assumes that the operands to the expression are integers and are literal values (not variables). When executed, the program repeatedly accepts and evaluates postfix expressions until the user chooses not to.

LISTING 3.2

```
/**
 * @author Lewis and Chase
 *
 * Demonstrates the use of a stack to evaluate postfix expressions.
 */

import java.util.Scanner;

public class Postfix
{
  /**
   * Reads and evaluates multiple postfix expressions.
   */

  public static void main (String[] args)
  {
    String expression, again;
    int result;

    try
    {
      Scanner in = new Scanner(System.in);
```

LISTING 3.2 *continued*

```
      do
      {
        PostfixEvaluator evaluator = new PostfixEvaluator();
        System.out.println ("Enter a valid postfix expression: ");
        expression = in.nextLine();

        result = evaluator.evaluate (expression);
        System.out.println();
        System.out.println ("That expression equals " + result);

        System.out.print ("Evaluate another expression [Y/N]? ");
        again = in.nextLine();
        System.out.println();
      }
      while (again.equalsIgnoreCase("y"));
    }
    catch (Exception IOException)
  {
    System.out.println(“Input exception reported");
    }
  }
}
```

LISTING 3.3

```
/**
 * PostfixEvaluator.java       Authors: Lewis/Chase
 *
 * Represents an integer evaluator of postfix expressions. Assumes
 * the operands are constants.
 */
import jss2.ArrayStack;
import java.util.StringTokenizer;

public class PostfixEvaluator
{
  /** constant for addition symbol */
  private final char ADD = '+';
  /** constant for subtraction symbol */
  private final char SUBTRACT = '-';
  /** constant for multiplication symbol */
```

LISTING 3.3 *continued*

```
private final char MULTIPLY = '*';
/** constant for division symbol */
private final char DIVIDE = '/';
/** the stack */
private ArrayStack<Integer> stack;

/**
 * Sets up this evaluator by creating a new stack.
 */
public PostfixEvaluator()
{
 stack = new ArrayStack<Integer>();
}

/**
 * Evaluates the specified postfix expression. If an operand is
 * encountered, it is pushed onto the stack. If an operator is
 * encountered, two operands are popped, the operation is
 * evaluated, and the result is pushed onto the stack.
 * @param expr String representation of a postfix expression
 * @return int value of the given expression
 */
public int evaluate (String expr)
{
  int op1, op2, result = 0;
  String token;
  StringTokenizer tokenizer = new StringTokenizer (expr);

  while (tokenizer.hasMoreTokens())
  {
    token = tokenizer.nextToken();

    if (isOperator(token))
    {
      op2 = (stack.pop()).intValue();
      op1 = (stack.pop()).intValue();
      result = evalSingleOp (token.charAt(0), op1, op2);
      stack.push (new Integer(result));
    }
    else
      stack.push (new Integer(Integer.parseInt(token)));
  }
  return result;
}
```

LISTING 3.3 *continued*

```
  /**
   * Determines if the specified token is an operator.
   * @param token String representing a single token
   * @return boolean true if token is operator
   */
  private boolean isOperator (String token)
  {
    return ( token.equals("+") || token.equals("-") ||
             token.equals("*") || token.equals("/") );
  }

  /**
   * Performs integer evaluation on a single expression consisting of
   * the specified operator and operands.
   * @param operation operation to be performed
   * @param op1 the first operand
   * @param op2 the second operand
   * @return int value of the expression
   */
  private int evalSingleOp (char operation, int op1, int op2)
  {
    int result = 0;

    switch (operation)
    {
    case ADD:
      result = op1 + op2;
      break;
    case SUBTRACT:
      result = op1 - op2;
      break;
    case MULTIPLY:
      result = op1 * op2;
      break;
    case DIVIDE:
      result = op1 / op2;
    }
    return result;
  }
}
```

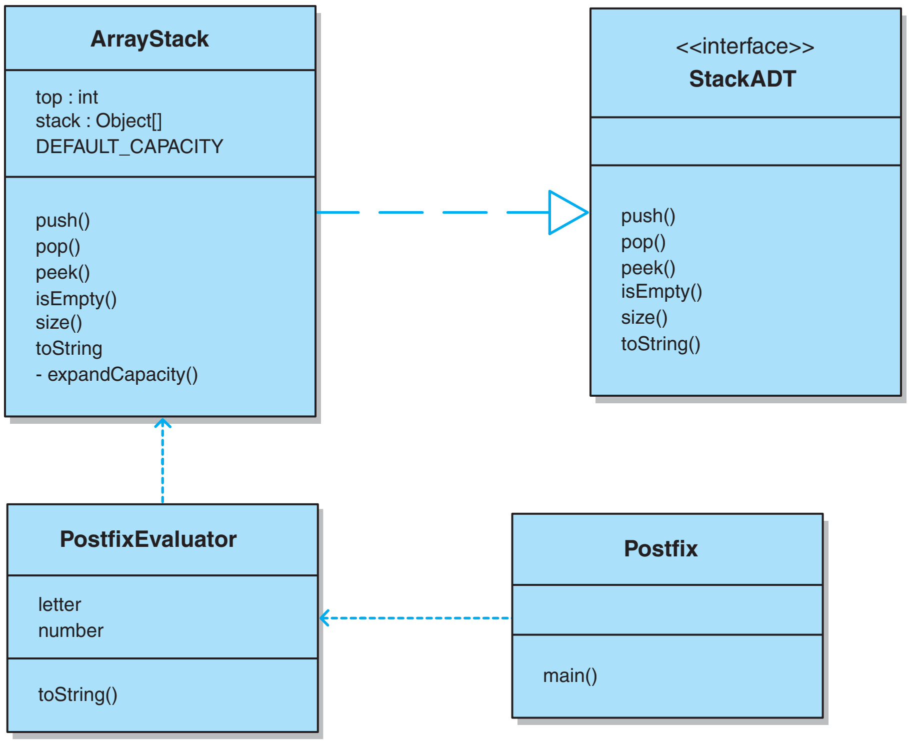


FIGURE 3.8 UML class diagram for the postfix expression evaluation program

The `evaluate` method performs the evaluation algorithm described earlier, supported by the `isOperator` and `evalSingleOp` methods. Note that in the `evaluate` method, only operands are pushed onto the stack. Operators are used as they are encountered and are never put on the stack. This is consistent with the evaluation algorithm we discussed. An operand is put on the stack as an `Integer` object, instead of as an `int` primitive value, because the stack data structure is designed to store objects.

When an operator is encountered, the most recent two operands are popped off of the stack. Note that the first operand popped is actually the second operand in the expression, and the second operand popped is the first operand in the expression. This order doesn't matter in the cases of addition and multiplication, but it certainly matters for subtraction and division.

Note also that the postfix expression program assumes that the postfix expression entered is valid, meaning that it contains a properly organized set of operators and operands. A postfix expression is invalid if either (1) two operands are not available on the stack when an operator is encountered or (2) there is more than one value on the stack when the tokens in the expression are exhausted. Either situation indicates that there was something wrong with the format of the expression, and both can be caught by examining the state of the stack at the appropriate point in the program. We will discuss how we might deal with these situations and other exceptional cases in the next section.

Perhaps the most important aspect of this program is the use of the class that defined the stack collection. At this point, we don't know how the stack was implemented. We simply trusted the class to do its job. In this example, we used the class `ArrayStack`, but we could have used any class that implemented a stack as long as it performed the stack operations (defined by the `StackADT` interface) as expected. From the point of view of evaluating postfix expressions, the manner in which the stack is implemented is largely irrelevant. Figure 3.8 shows a UML class diagram for the postfix expression evaluation program.

3.6 Exceptions

One concept that we will explore with each of the collections we discuss is that of exceptional behavior. What action should the collection take in the exceptional case? There are some such cases that are inherent in the collection itself. For example, in the case of a stack, what should happen if an attempt is made to pop an element from an empty stack? In this case, it does not matter what data structure is being used to implement the collection; the exception will still apply. Some such cases are artifacts of the data structure being used to implement the collection. For example, if we are using an array to implement a stack, what should happen if an attempt is made to push an element onto the stack but the array is full? Let's take a moment to explore this concept further.

Problems that arise in a Java program may generate exceptions or errors. An *exception* is an object that defines an unusual or erroneous situation. An exception is thrown by a program or the run-time environment, and it can be caught and handled appropriately if desired. An *error* is similar to an exception, except that an error generally represents an unrecoverable situation, and it should not be caught. Java has a predefined set of exceptions and errors that may occur during the execution of a program.

In our postfix evaluation example, there were several potential exceptional situations. For example:

- If the stack were full on a push
- If the stack were empty on a pop
- If the stack held more than one value at the completion of the evaluation

Let's consider each of these separately. The possibility that the stack might be full on a push is an issue for the underlying data structure, not the collection. Conceptually speaking, there is no such thing as a full stack. Now we know that this is not reality and that all data structures will eventually reach a limit. However, even when this physical limit is reached, the stack is not full; only the data structure that implements the stack is full. We will discuss strategies for handling this situation as we implement our stack in the next section.

What if the stack is empty on a pop? This is an exceptional case that has to do with the problem and not the underlying data structure. In our postfix evaluation example, if we attempt to pop two operands and there are not two operands available on the stack, our postfix expression was not properly formed. This is a perfect case where the collection needs to report the exception and the application then must interpret that exception in context.

KEY CONCEPT
Errors and exceptions represent unusual or invalid processing.

The third case is equally interesting. What if the stack holds more than one value at the completion of the evaluation? From the perspective of the stack collection, this is not an exception. However, from the perspective of the application, this is a problem that means once again that the postfix expression was not well formed. Because it will not generate an exception from the collection, this is a condition for which the application must test.

A program can be designed to process an exception in one of three ways:

- Not handle the exception at all.
- Handle the exception where it occurs.
- Handle the exception at another point in the program.

We explore each of these approaches in the following sections.

Exception Messages

If an exception is not handled at all by the program, the program will terminate (abnormally) and produce a message that describes what exception occurred and where in the program it was produced. The information associated with an exception is often helpful in tracking down the cause of a problem.

Let's look at the output of an exception. An `ArithmeticException` is thrown when an invalid arithmetic operation is attempted, such as dividing by zero. When that exception is thrown, if there is no code in the program to handle the exception explicitly, the program terminates and prints a message similar to the following:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Zero.main (Zero.java:17)
```

The first line of the exception output indicates which exception was thrown and provides some information about why it was thrown. The remaining line or lines are the *call stack trace*, which indicates where the exception occurred. In this case, there is only one line in the call stack trace, but there may be several, depending on where the exception originated in the program. The first line of the trace indicates the method, file, and line number where the exception occurred. The other lines in the trace, if present, indicate the methods that were called to get to the method that produced the exception. In this program, there is only one method, and it produced the exception; therefore, there is only one line in the trace.

The call stack trace information is also available by calling methods of the exception object that is being thrown. The method `getMessage` returns a string explaining the reason the exception was thrown. The method `printStackTrace` prints the call stack trace.

KEY CONCEPT

The messages printed by a thrown exception indicate the nature of the problem and provide a method call stack trace.

The `try` Statement

Let's now examine how we catch and handle an exception when it is thrown. A *try statement* consists of a `try` block followed by one or more `catch` clauses. The `try` block is a group of statements that may throw an exception. A `catch` clause defines how a particular kind of exception is handled. A `try` block can have several `catch` clauses associated with it, each dealing with a particular kind of exception. A `catch` clause is sometimes called an *exception handler*.

Here is the general format of a `try` statement:

```
try
{
   // statements in the try block
}
catch (IOException exception)
{
   // statements that handle the I/O problem
}
catch (NumberFormatException exception)
{
   // statements that handle the number format problem
}
```

When a `try` statement is executed, the statements in the `try` block are executed. If no exception is thrown during the execution of the `try` block, processing continues with the statement following the `try` statement (after all of the `catch` clauses). This situation is the normal execution flow and should occur most of the time.

KEY CONCEPT

Each `catch` clause on a `try` statement handles a particular kind of exception that may be thrown within the `try` block.

If an exception is thrown at any point during the execution of the `try` block, control is immediately transferred to the appropriate exception handler if it is present. That is, control transfers to the first `catch` clause whose specified exception corresponds to the class of the exception that was thrown. After executing the statements in the `catch` clause, control is transferred to the statement after the entire `try` statement.

Exception Propagation

If an exception is not caught and handled where it occurs, control is immediately returned to the method that invoked the method that produced the exception. We can design our software so that the exception is caught and handled at this outer level. If it isn't caught there, control returns to the method that called it. This process is called *propagating the exception*.

KEY CONCEPT
If an exception is not caught and handled where it occurs, it is propagated to the calling method.

KEY CONCEPT
A programmer must carefully consider how exceptions should be handled, if at all, and at what level.

Exception propagation continues until the exception is caught and handled, or until it is propagated out of the `main` method, which terminates the program and produces an exception message. To catch an exception at an outer level, the method that produces the exception must be invoked inside a `try` block that has an appropriate `catch` clause to handle it.

A programmer must pick the most appropriate level at which to catch and handle an exception. There is no single best answer. It depends on the situation and the design of the system. Sometimes the right approach will be not to catch an exception at all and let the program terminate.

The manner in which exceptions are used is important to the definition of a software system. Exceptions could be thrown in many situations in a collection. Usually it's best to throw exceptions whenever an invalid operation is attempted. For example, in the case of a stack, we will throw an exception whenever the user attempts to pop an element from an empty stack. The user then has the choice of checking the situation beforehand to avoid the exception:

```
if (! theStack.isEmpty())
   element = theStack.pop();
```

Or the user can use a `try-catch` statement to handle the situation when it does occur:

```
try {
    element = theStack.pop()
}
catch (EmptyCollectionException exception)
{
   System.out.println ("No elements available.");
}
```

As we explore particular implementation techniques for a collection, we will also discuss the appropriate use of exceptions.

3.7 Implementing a Stack: With Arrays

So far in our discussion of a stack collection we have described its basic conceptual nature and the operations that allow the user to interact with it. In software engineering terms, we would say that we have done the analysis for a stack collection. We have also used a stack, without knowing the details of how it was implemented, to solve a particular problem. Now let's turn our attention to the implementation details. There are various ways to implement a class that represents a stack. In this section, we examine an implementation strategy that uses an array to store the objects contained in the stack. In the next chapter, we examine a second technique for implementing a stack.

KEY CONCEPT
The implementation of the collection operations should not affect the way users interact with the collection.

To explore this implementation, we must recall several key characteristics of Java arrays. The elements stored in an array are indexed from 0 to n–1, where n is the total number of cells in the array. An array is an object, which is instantiated separately from the objects it holds. And when we talk about an array of objects, we are actually talking about an array of references to objects, as pictured in Figure 3.9.

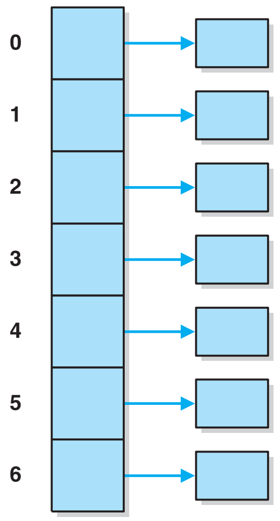


FIGURE 3.9 An array of object references

Keep in mind the separation between the collection and the underlying data structure used to implement it. Our goal is to design an efficient implementation that provides the functionality of every operation defined in the stack abstract data type. The array is just a convenient data structure in which to store the objects.

Managing Capacity

When an array object is created, it is allocated a specific number of cells into which elements can be stored. For example, the following instantiation creates an array that can store 500 elements, indexed from 0 to 499:

```
Object[] collection = Object[500];
```

The number of cells in an array is called its *capacity*. This value is stored in the `length` constant of the array. The capacity of an array cannot be changed once the array has been created.

When using an array to implement a collection, we have to deal with the situation in which all cells of the array are being used to store elements. That is, because we are using a fixed-size data structure, at some point the data structure may become "full." However, just because the data structure is full, should that mean that the collection is full?

A crucial question in the design of a collection is what to do in the case in which a new element is added to a full data structure. Three basic options exist:

- We could implement operations that add an element to the collection such that they throw an exception if the data structure is full.
- We could implement the `add` operations to return a status indicator that can be checked by the user to see if the `add` operation was successful.
- We could automatically expand the capacity of the underlying data structure whenever necessary so that, essentially, it would never become full.

KEY CONCEPT
How we handle exceptional conditions determines whether the collection or the user of the collection controls the particular behavior.

In the first two cases, the user of the collection must be aware that the collection could get full and take steps to deal with it when needed. For these solutions we would also provide extra operations that allow the user to check to see if the collection is full and to expand the capacity of the data structure as desired. The advantage of these approaches is that it gives the user more control over the capacity.

However, given our goal to separate the interface from the implementation, the third option is attractive. The capacity of the underlying data structure is an implementation detail that, in general, should be hidden from the user. Furthermore, the capacity issue is particular to this implementation. Other techniques used to implement the collection, such as the one we explore in the next chapter, are not restricted by a fixed capacity and therefore never have to deal with this issue.

In the solutions presented in this book, we opt to implement fixed data structure solutions by automatically expanding the capacity of the underlying data structure. Occasionally, other options are explored as programming projects.

3.8 The `ArrayStack` Class

In the Java Collections API framework, class names indicate both the underlying data structure and the collection. We follow that naming convention in this book. Thus, we define a class called `ArrayStack` to represent a stack with an underlying array-based implementation.

To be more precise, we define a class called `ArrayStack<T>` that represents an array-based implementation of a stack collection that stores objects of generic type `T`. When we instantiate an `ArrayStack` object, we specify what the generic type `T` represents.

An array implementation of a stack can be designed by making the following four assumptions: the array is an array of object references (type determined when the stack is instantiated), the bottom of the stack is always at index 0 of the array, the elements of the stack are stored in order and contiguously in the array, and there is an integer variable `top` that stores the index of the array immediately following the top element in the stack.

KEY CONCEPT
For efficiency, an array-based stack implementation keeps the bottom of the stack at index 0.

Figure 3.10 illustrates this configuration for a stack that currently contains the elements A, B, C, and D, assuming that they have been pushed on in that order. To simplify the figure, the elements are shown in the array itself rather than as objects referenced from the array. Note that the variable top represents both the next cell into which a pushed element should be stored as well as the count of the number of elements currently in the stack.

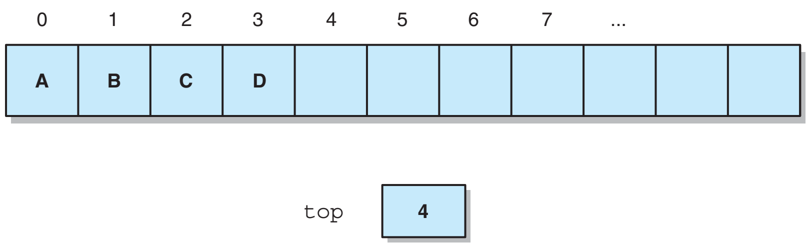


FIGURE 3.10 An array implementation of a stack

In this implementation, the bottom of the stack is always held at index 0 of the array, and the stack grows and shrinks at the higher indexes. This is considerably more efficient than if the stack were reversed within the array. Consider the processing that would be necessary if the top of the stack were kept at index 0.

From these assumptions, we can determine that our class will need a constant to store the default capacity, a variable to keep track of the top of the stack, and a variable for the array to store the stack. This results in the following class header. Note that our `ArrayStack` class will be part of the `jss2` package and will make use of a package called `jss2.exceptions`.

```java
/**
 * @author Lewis and Chase
 *
 * Represents an array implementation of a stack.
 */
package jss2;
import jss2.exceptions.*;

public class ArrayStack<T> implements StackADT<T>
{
  /**
   * constant to represent the default capacity of the array
   */
  private final int DEFAULT_CAPACITY = 100;
  /**
   * int that represents both the number of elements and the next
   * available position in the array
   */

  private int top;
  /**
   * array of generic elements to represent the stack
   */
  private T[] stack;
```

The Constructors

Our class will have two constructors, one to use the default capacity and the other to use a specified capacity.

```
/**
 * Creates an empty stack using the default capacity.
 */

public ArrayStack()
{

  top = 0;
  stack = (T[])(new Object[DEFAULT_CAPACITY]);
}
/**
 * Creates an empty stack using the specified capacity.
 * @param initialCapacity represents the specified capacity
 */
public ArrayStack (int initialCapacity)
{
  top = 0;
  stack = (T[])(new Object[initialCapacity]);
}
```

Just to refresh our memory, this is an excellent example of method overloading (i.e., two methods with the same name that differ only in the parameter list).

From our previous discussion of generics we recall that you cannot instantiate a generic type. This also means that you cannot instantiate an array of a generic type. This results in an interesting line of code in both of our constructors:

```
stack = (T[])(new Object[DEFAULT_CAPACITY]);
```

Note that in this line, we are instantiating an array of Objects and then casting it as an array of our generic type. This will create a compile-time warning for an unchecked type conversion because the Java compiler cannot guarantee the type safety of this cast. As we have seen from our earlier discussion, it is worth this warning to gain the flexibility and safety of generics.

The push Operation

To push an element onto the stack, we simply insert it in the next available position in the array as specified by the variable `top`. Before doing so, however, we must determine if the array has reached its capacity and expand it if necessary. After storing the value, we must update the value of `top` so that it continues to represent the number of elements in the stack.

Implementing these steps results in the following code:

```
/**
 * Adds the specified element to the top of this stack, expanding
 * the capacity of the stack array if necessary.

 * @param element generic element to be pushed onto stack
 */
public void push (T element)
{
   if (size() == stack.length)
      expandCapacity();

   stack[top] = element;
   top++;
}
```

The `expandCapacity` method is implemented to double the size of the array as needed. Of course, since an array cannot be resized once it is instantiated, this method simply creates a new larger array and then copies the contents of the old array into the new one. It serves as a support method of the class and can therefore be implemented with private visibility.

```
/**
 * Creates a new array to store the contents of this stack with
 * twice the capacity of the old one.
 */
private void expandCapacity()
{
   T[] larger = (T[])(new Object[stack.length*2]);

   for (int index=0; index < stack.length; index++)
      larger[index] = stack[index];

   stack = larger;
}
```

Figure 3.11 illustrates the result of pushing an element E onto the stack that was depicted in Figure 3.10.

The `push` operation for the array implementation of a stack consists of the following steps:

- Make sure that the array is not full.
- Set the reference in position `top` of the array to the object being added to the stack.
- Increment the values of `top` and `count`.

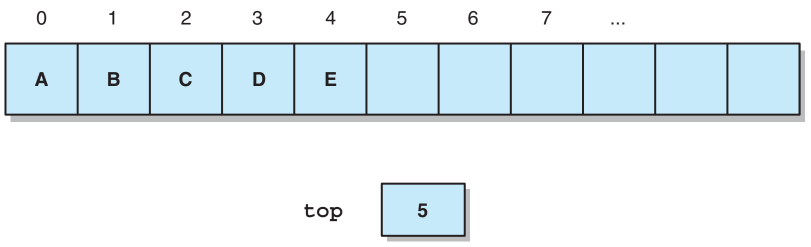


FIGURE 3.11 The stack after pushing element E

Each of these steps is O(1). Thus the operation is O(1). We might wonder about the time complexity of the `expandCapacity` method and the impact it might have on the analysis of the `push` method. This method does contain a linear `for` loop and, intuitively, we would call that O(n). However, given how seldom the `expandCapacity` method is called relative to the number of times `push` may be called, we can amortize that complexity across all instances of `push`.

The pop Operation

The `pop` operation removes and returns the element at the top of the stack. For an array implementation, that means returning the element at index top - 1. Before attempting to return an element, however, we must ensure that there is at least one element in the stack to return.

The array-based version of the `pop` operation can be implemented as follows:

```
/**
 * Removes the element at the top of this stack and returns a
 * reference to it. Throws an EmptyCollectionException if the stack
 * is empty.
 * @return T element removed from top of stack
 * @throws EmptyCollectionException if a pop is attempted on empty stack
 */
public T pop() throws EmptyCollectionException
{
   if (isEmpty())
      throw new EmptyCollectionException("Stack");

   top--;
   T result = stack[top];
   stack[top] = null;

   return result;
}
```

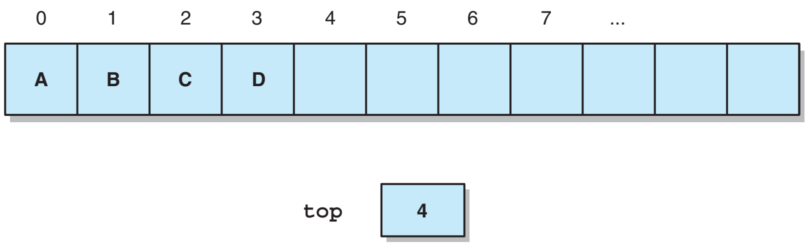


FIGURE 3.12 The stack after popping the top element

If the stack is empty when the `pop` method is called, an `EmptyCollectionException` is thrown. Otherwise, the value of `top` is decremented and the element stored at that location is stored into a temporary variable so that it can be returned. That cell in the array is then set to null. Note that the value of `top` ends up with the appropriate value relative to the now smaller stack. Figure 3.12 illustrates the results of a `pop` operation on the stack from Figure 3.10, which brings it back to its earlier state (identical to Figure 3.10).

The `pop` operation for the array implementation consists of the following steps:

- Make sure the stack is not empty.
- Decrement the `top` counter.
- Set a temporary reference equal to the element in `stack[top]`.
- Set `stack[top]` equal to null.
- Return the temporary reference.

All of these steps are also O(1). Thus, the `pop` operation for the array implementation has time complexity O(1).

The peek Operation

The `peek` operation returns a reference to the element at the top of the stack without removing it from the array. For an array implementation, that means returning a reference to the element at position top-1. This one step is O(1) and thus the `peek` operation is O(1) as well.

```
/**
 * Returns a reference to the element at the top of this stack.
 * The element is not removed from the stack. Throws an
 * EmptyCollectionException if the stack is empty.
 * @return T element on top of stack
 * @throws EmptyCollectionException if a peek is attempted on empty stack
 */
public T peek() throws EmptyCollectionException
{
   if (isEmpty())
      throw new EmptyCollectionException("Stack");

   return stack[top-1];
}
```

Other Operations

The `isEmpty`, `size`, and `toString` operations and their analysis are left as programming projects exercises, respectively.

Summary of Key Concepts

- A collection is an object that gathers and organizes other objects.
- Elements in a collection are typically organized by the order of their addition to the collection or by some inherent relationship among the elements.
- A collection is an abstraction where the details of the implementation are hidden.
- A data structure is the underlying programming construct used to implement a collection.
- Stack elements are processed in a LIFO manner—the last element in is the first element out.
- A programmer should choose the structure that is appropriate for the type of data management needed.
- Inheritance is the process of deriving a new class from an existing one.
- One purpose of inheritance is to reuse existing software.
- Inherited variables and methods can be used in the derived class as if they had been declared locally.
- Inheritance creates an is-a relationship between all parent and child classes.
- The child of one class can be the parent of one or more other classes, creating a class hierarchy.
- Common features should be located as high in a class hierarchy as is reasonable, minimizing maintenance efforts.
- All Java classes are derived, directly or indirectly, from the `Object` class.
- The `toString` and `equals` methods are defined in the `Object` class and therefore are inherited by every class in every Java program.
- A polymorphic reference can refer to different types of objects over time.
- A reference variable can refer to any object created from any class related to it by inheritance.
- A polymorphic reference uses the type of the object, not the type of the reference, to determine which version of a method to invoke.
- A Java interface defines a set of abstract methods and is useful in separating the concept of an abstract data type from its implementation.
- By using the interface name as a return type, the interface doesn't commit the method to the use of any particular class that implements a stack.
- A stack is the ideal data structure to use when evaluating a postfix expression.

- Errors and exceptions represent unusual or invalid processing.
- The messages printed by a thrown exception indicate the nature of the problem and provide a method call stack trace.
- Each catch clause on a try statement handles a particular kind of exception that may be thrown within the try block.
- If an exception is not caught and handled where it occurs, it is propagated to the calling method.
- A programmer must carefully consider how exceptions should be handled, if at all, and at what level.
- The implementation of the collection operations should not affect the way users interact with the collection.
- How we handle exceptional conditions determines whether the collection or the user of the collection controls the particular behavior.
- For efficiency, an array-based stack implementation keeps the bottom of the stack at index 0.

Self-Review Questions

SR 3.1 What is a collection?

SR 3.2 What is a data type?

SR 3.3 What is an abstract data type?

SR 3.4 What is a data structure?

SR 3.5 What is abstraction and what advantage does it provide?

SR 3.6 Why is a class an excellent representation of an abstract data type?

SR 3.7 What is the characteristic behavior of a stack?

SR 3.8 What are the five basic operations on a stack?

SR 3.9 What are some of the other operations that might be implemented for a stack?

SR 3.10 Define inheritance.

SR 3.11 Define polymorphism.

SR 3.12 Given the example in Figure 3.5, list the subclasses of Mammal.

SR 3.13 Given the example in Figure 3.5, will the following code compile?

```
Animal creature = new Parrot();
```

SR 3.14 Given the example in Figure 3.5, will the following code compile?

```
Horse creature = new Mammal();
```

SR 3.15 What is the purpose of Generics in the Java language?

SR 3.16 What is the advantage of postfix notation?

Exercises

EX 3.1 Compare and contrast data types, abstract data types, and data structures.

EX 3.2 List the collections in the Java Collections API and mark the ones that are covered in this text.

EX 3.3 Define the concept of abstraction and explain why it is important in software development.

EX 3.4 Hand trace a stack X through the following operations:

```
X.push(new Integer(4));
X.push(new Integer(3));
Integer Y = X.pop();
X.push(new Integer(7));
X.push(new Integer(2));
X.push(new Integer(5));
X.push(new Integer(9));
Integer Y = X.pop();
X.push(new Integer(3));
X.push(new Integer(9));
```

EX 3.5 Given the resulting stack X from the previous exercise, what would be the result of each of the following?

```
a. Y = X.peek();
b. Y = X.pop();
   Z = X.peek();
c. Y = X.pop();
   Z = X.peek();
```

EX 3.6 What should be the time complexity of the `isEmpty()`, `size()`, and `toString()` methods?

EX 3.7 Show how the undo operation in a word processor can be supported by the use of a stack. Give specific examples and draw the contents of the stack after various actions are taken.

EX 3.8 In the postfix expression evaluation example, the two most recent operands are popped when an operator is encountered so that the subexpression can be evaluated. The first operand popped is treated as the second operand in the subexpression, and the second operand popped is the first. Give and explain an example that demonstrates the importance of this aspect of the solution.

EX 3.9 Draw an example using the five integers (12, 23, 1, 45, 9) of how a stack could be used to reverse the order (9, 45, 1, 23, 12) of these elements.

EX 3.10 Explain what would happen to the algorithms and the time complexity of an array implementation of the stack if the top of the stack were at position 0.

Programming Projects

PP 3.1 Complete the implementation of the `ArrayStack` class presented in this chapter. Specifically, complete the implementations of the `peek`, `isEmpty`, `size`, and `toString` methods.

PP 3.2 Design and implement an application that reads a sentence from the user and prints the sentence with the characters of each word backwards. Use a stack to reverse the characters of each word.

PP 3.3 Modify the solution to the postfix expression evaluation problem so that it checks for the validity of the expression that is entered by the user. Issue an appropriate error message when an erroneous situation is encountered.

PP 3.4 The array implementation in this chapter keeps the top variable pointing to the next array position above the actual top of the stack. Rewrite the array implementation such that `stack[top]` is the actual top of the stack.

PP 3.5 There is a data structure called a drop-out stack that behaves like a stack in every respect except that if the stack size is n, when the n+1 element is pushed, the first element is lost. Implement a drop-out stack using an array. (*Hint:* a circular array implementation would make sense.)

PP 3.6 Implement an integer adder using three stacks.

PP 3.7 Implement an infix-to-postfix translator using stacks.

PP 3.8 Implement a class called reverse that uses a stack to output a set of elements input by the user in reverse order.

PP 3.9 Create a graphical application that provides a button for push and pop from a stack, a text field to accept a string as input for push, and a text area to display the contents of the stack after each operation.

Answers to Self-Review Questions

SRA 3.1 A collection is an object that gathers and organizes other objects.

SRA 3.2 A data type is a set of values and operations on those values defined within a programming language.

SRA 3.3 An abstract data type is a data type that is not defined within the programming language and must be defined by the programmer.

SRA 3.4 A data structure is the set of objects necessary to implement an abstract data type.

SRA 3.5 Abstraction is the concept of hiding the underlying implementation of operations and data storage in order to simplify the use of a collection.

SRA 3.6 Classes naturally provide abstraction since only those methods that provide services to other classes have public visibility.

SRA 3.7 A stack is a last in, first out (LIFO) structure.

SRA 3.8 The operations are:

push—adds an element to the end of the stack

pop—removes an element from the front of the stack

peek—returns a reference to the element at the front of the stack

isEmpty—returns true if the stack is empty, returns false otherwise

size—returns the number of elements in the stack

SRA 3.9 `makeEmpty(), destroy(), full()`

SRA 3.10 Inheritance is the process in which a new class is derived from an existing one. The new class automatically contains some or all of the variables and methods in the original class. Then, to tailor the class as needed, the programmer can add new variables and methods to the derived class, or modify the inherited ones.

SRA 3.11 The term *polymorphism* can be defined as "having many forms." A *polymorphic reference* is a reference variable that can refer to different types of objects at different points in time. The specific method invoked through a polymorphic reference can change from one invocation to the next.

SRA 3.12 The subclasses of `Mammal` are `Horse` and `Bat`.

SRA 3.13 Yes, a reference variable of a parent class or any superclass may hold a reference to one of its descendants.

SRA 3.14 No, a reference variable for a child or subclass may not hold a reference to a parent or superclass. To make this assignment, you would have to explicitly cast the parent class into the child class (`Horse creature = (Horse)(new Mammal());`

SRA 3.15 Beginning with Java 5.0, Java enables us to define a class based on a *generic type*. That is, we can define a class so that it stores, operates on, and manages objects whose type is not specified until the class is instantiated. This allows for the creation of structures that can manipulate "generic" elements and still provide type checking.

SRA 3.16 Postfix notation avoids the need for precedence rules that are required to evaluate infix expressions.

This page intentionally left blank

Linked Structures

4

This chapter explores a technique for creating data structures using references to create links between objects. Linked structures are fundamental in the development of software, especially the design and implementation of collections. This approach has both advantages and disadvantages when compared to a solution using arrays.

CHAPTER OBJECTIVES

- Describe the use of references to create linked structures
- Compare linked structures to array-based structures
- Explore the techniques for managing a linked list
- Discuss the need for a separate node object to form linked structures
- Implement a stack collection using a linked list

4.1 References as Links

KEY CONCEPT
Object reference variables can be used to create linked structures.

In Chapter 3, we discussed the concept of collections and explored one collection in particular: a stack. We defined the operations on a stack collection and designed an implementation using an underlying array-based data structure. In this chapter, we explore an entirely different approach to designing a data structure.

A *linked structure* is a data structure that uses object reference variables to create links between objects. Linked structures are the primary alternative to an array-based implementation of a collection. After discussing various issues involved in linked structures, we will define a new implementation of a stack collection that uses an underlying linked data structure and demonstrate its use.

Recall that an object reference variable holds the address of an object, indicating where the object is stored in memory. The following declaration creates a variable called `obj` that is only large enough to hold the numeric address of an object:

```
Object obj;
```

Usually the specific address that an object reference variable holds is irrelevant. That is, while it is important to be able to use the reference variable to access an object, the specific location in memory where it is stored is unimportant. Therefore, instead of showing addresses, we usually depict a reference variable as a name that "points to" an object, as shown in Figure 4.1. A reference variable, used in this context, is sometimes called a *pointer*.

Consider the situation in which a class defines as instance data a reference to another object of the same class. For example, suppose we have a class named `Person` that contains a person's name, address, and other relevant information. Now suppose that in addition to this data, the `Person` class also contains a reference variable to another `Person` object:

```
public class Person
{
   private String name;
   private String address;

   private Person next;  // a link to another Person object

   // whatever else
}
```

Using only this one class, a linked structure can be created. One `Person` object contains a link to a second `Person` object. This second object also contains a reference to a `Person`, which contains another, and so on. This type of object is sometimes called *self-referential*.

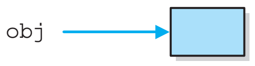


FIGURE 4.1 An object reference variable pointing to an object

This kind of relationship forms the basis of a *linked list*, which is a linked structure in which one object refers to the next, creating a linear ordering of the objects in the list. A linked list is depicted in Figure 4.2. Often the objects stored in a linked list are referred to generically as the *nodes* of the list.

Note that a separate reference variable is needed to indicate the first node in the list. The list is terminated in a node whose `next` reference is `null`.

KEY CONCEPT

A linked list is composed of objects that each point to the next object in the list.

A linked list is only one kind of linked structure. If a class is set up to have multiple references to objects, a more complex structure can be created, such as the one depicted in Figure 4.3. The way in which the links are managed dictates the specific organization of the structure.

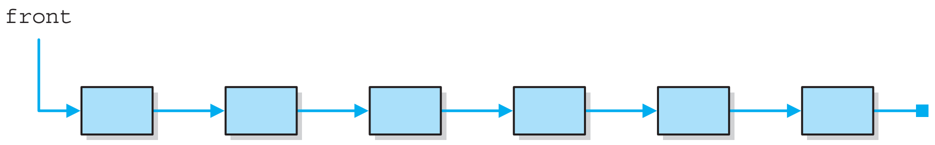


FIGURE 4.2 A linked list

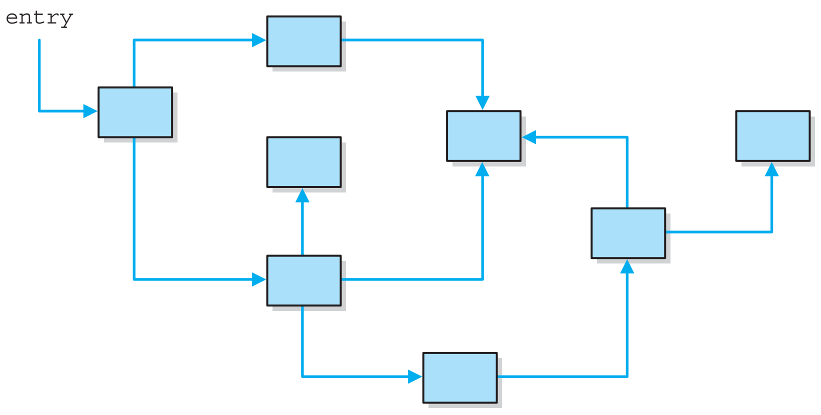


FIGURE 4.3 A complex linked structure

KEY CONCEPT

A linked list dynamically grows as needed and essentially has no capacity limitations.

For now, we will focus on the details of a linked list. Many of these techniques apply to more complicated linked structures as well.

Unlike an array, which has a fixed size, a linked list has no upper bound on its capacity other than the limitations of memory in the computer. A linked list is considered to be a *dynamic* structure because its size grows and shrinks as needed to accommodate the number of elements stored. In Java, all objects are created dynamically from an area of memory called the *system heap*, or *free store*.

The next section explores some of the primary ways in which a linked list is managed.

4.2 Managing Linked Lists

Keep in mind that our goal is to use linked lists and other linked structures to create collections. Because the principal purpose of a collection is to be able to add, remove, and access elements, we must first examine how to accomplish these fundamental operations using links.

No matter what a linked list is used to store, there are a few basic techniques involved in managing the nodes in the list. Specifically, elements in the list are accessed, elements are inserted into a list, and elements are removed from the list.

Accessing Elements

Special care must be taken when dealing with the first node in the list so that the reference to the entire list is maintained appropriately. When using linked lists, we maintain a pointer to the first element in the list. To access other elements, we must access the first one and then follow the next pointer from that one to the next one and so on. Consider our previous example of a `person` class containing the attributes `name`, `address`, and `next`. If we wanted to find the fourth person in the list, and assuming that we had a variable `first` of type `Person` that pointed to the first person in the list and that the list contained at least four nodes, we might use the following code:

```
Person current = first;
for (int i = 0, i < 3, i++)
    current = current.next;
```

After executing this code, `current` will point to the fourth person in the list. Notice that it is very important to create a new reference variable, in this case `current`, and then start by setting that reference variable to point to the first element in the list. Consider what would happen if we used the `first` pointer in the loop instead of `current`. Once we moved the `first` pointer to point to the second element in the list, we would no longer have a pointer to the first element and would not be able to access it. Keep in mind that with a linked list, the only way to access the elements in the list is to start with the first element and progress through the list.

Of course, a more likely scenario would be that we would need to search our list for a particular person. Assuming that the `Person` class overrides the `equals` method such that it returns true if the given `String` matches the `name` stored for that person, then the following code will search the list for Tom Jones:

```
String searchstring = "Tom Jones";
Person current = first;
while ((not(current.equals(searchstring)) && (current.next != null))
    current = current.next;
```

Note that this loop will terminate when the string is found or when the end of the list is encountered. Now that we have seen how to access elements in a linked list, let's consider how to insert elements into a list.

Inserting Nodes

A node may be inserted into a linked list at any location: at the front of the list, among the interior nodes in the middle of the list, or at the end of the list. Adding a node to the front of the list requires resetting the reference to the entire list, as shown in Figure 4.4. First, the `next` reference of the added node is set to point to the current first node in the list. Second, the reference to the front of the list is reset to point to the newly added node.

KEY CONCEPT

The order in which references are changed is crucial to maintaining a linked list.

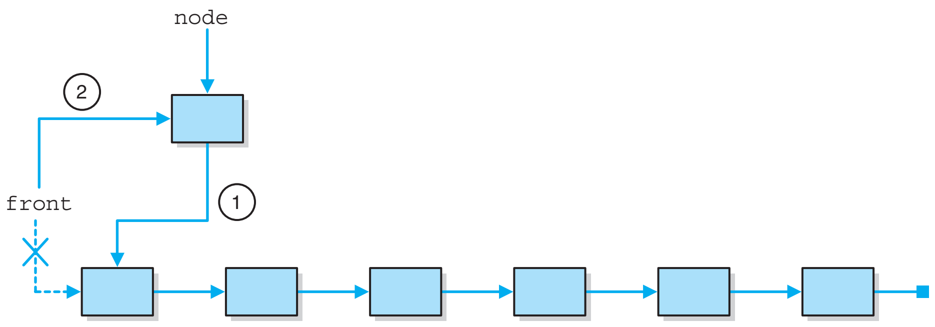


FIGURE 4.4 Inserting a node at the front of a linked list

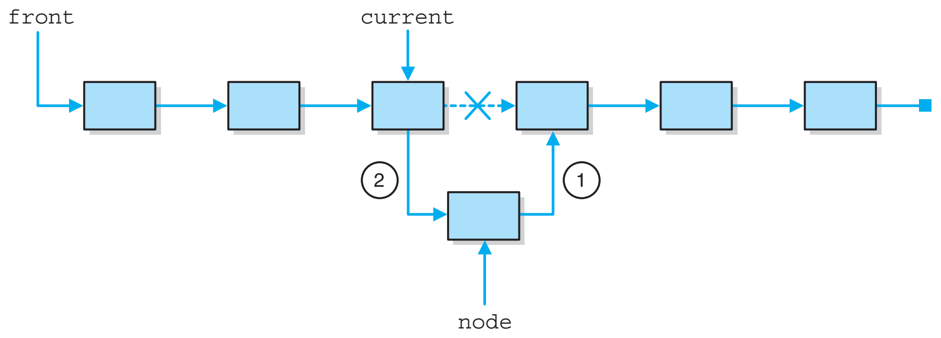


FIGURE 4.5 Inserting a node in the middle of a linked list

Note that difficulties would arise if these steps were reversed. If we were to reset the `front` reference first, we would lose the only reference to the existing list and it could not be retrieved.

Inserting a node into the middle of a list requires some additional processing. First, we have to find the node in the list that will immediately precede the new node being inserted. Unlike an array, in which we can access elements using subscripts, a linked list requires that we use a separate reference to move through the nodes of the list until we find the one we want. This type of reference is often called `current`, because it indicates the current node in the list that is being examined.

Initially, `current` is set to point to the first node in the list. Then a loop is used to move the `current` reference along the list of nodes until the desired node is found. Once it is found, the new node can be inserted, as shown in Figure 4.5.

First, the `next` reference of the new node is set to point to the node *following* the one to which `current` refers. Then, the `next` reference of the current node is reset to point to the new node. Once again, the order of these steps is important.

This process will work wherever the node is to be inserted along the list, including making it the new second node in the list or making it the last node in the list. If the new node is inserted immediately after the first node in the list, then `current` and `front` will refer to the same (first) node. If the new node is inserted at the end of the list, the `next` reference of the new node is set to `null`. The only special case occurs when the new node is inserted as the first node in the list.

Deleting Nodes

KEY CONCEPT

Dealing with the first node in a linked list often requires special handling.

Any node in the list can be deleted. We must maintain the integrity of the list no matter which node is deleted. As with the process of inserting a node, dealing with the first node in the list represents a special case.

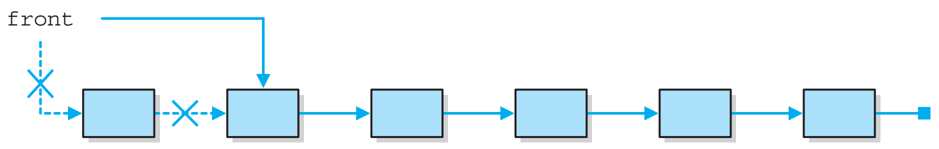


FIGURE 4.6 Deleting the first node in a linked list

To delete the first node in a linked list, we reset the reference to the front of the list so that it points to the current second node in the list. This process is shown in Figure 4.6. If the deleted node is needed elsewhere, a separate reference to it must be set up before resetting the `front` reference.

To delete a node from the interior of the list, we must first find the node *in front of* the node that is to be deleted. This processing often requires the use of two references: one to find the node to be deleted and another to keep track of the node immediately preceding that one. Thus, they are often called `current` and `previous`, as shown in Figure 4.7.

Once these nodes have been found, the `next` reference of the previous node is reset to point to the node pointed to by the `next` reference of the current node. The deleted node can then be used as needed.

Sentinel Nodes

Thus far, we have described insertion into and deletion from a list as having two cases: the case when dealing with the first node and the case when dealing with any other node. It is possible to eliminate the special case involving the first node by introducing a *sentinel node* or *dummy node* at the front of the list. A sentinel node serves as a false first node and doesn't actually represent an element in the list. By using a sentinel node, all insertions and deletions will fall under the second case and the implementations will not have as many special situations to consider.

KEY CONCEPT
Implementing a list with a sentinel node or dummy node as the first node eliminates the special cases dealing with the first node.

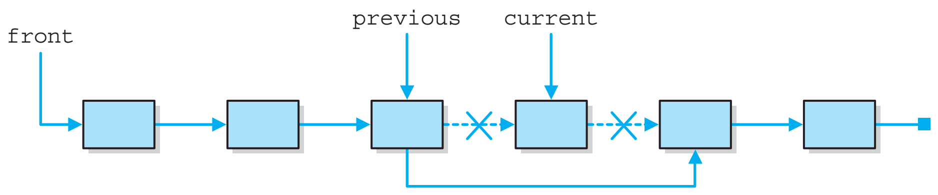


FIGURE 4.7 Deleting an interior node from a linked list

4.3 Elements Without Links

Now that we have explored some of the techniques needed to manage the nodes of a linked list, we can turn our attention to using a linked list as an alternative implementation approach for a collection. However, to do so we need to carefully examine one other key aspect of linked lists. We must separate the details of the linked list structure from the elements that the list stores.

KEY CONCEPT
Objects that are stored in a collection should not contain any implementation details of the underlying data structure.

Earlier in this chapter we discussed the idea of a `Person` class that contains, among its other data, a link to another `Person` object. The flaw in this approach is that the self-referential `Person` class must be designed so that it "knows" it may become a node in a linked list of `Person` objects. This assumption is impractical, and it violates our goal of separating the implementation details from the parts of the system that use the collection.

The solution to this problem is to define a separate node class that serves to link the elements together. A node class is fairly simple, containing only two important references: one to the next node in the linked list and another to the element that is being stored in the list. This approach is depicted in Figure 4.8.

The linked list of nodes can still be managed using the techniques discussed in the previous section. The only additional aspect is that the actual elements stored in the list are accessed using a separate reference in the node objects.

Doubly Linked Lists

An alternative implementation for linked structures is the concept of a doubly linked list, as illustrated in Figure 4.9. In a doubly linked list, two references are maintained: one to point to the first node in the list and another to point to the last node in the list. Each node in the list stores both a reference to the next element and a reference to the previous one. If we were to use sentinel nodes with a doubly linked list, we would place sentinel nodes on both ends of the list. We discuss doubly linked lists further in Chapter 6.

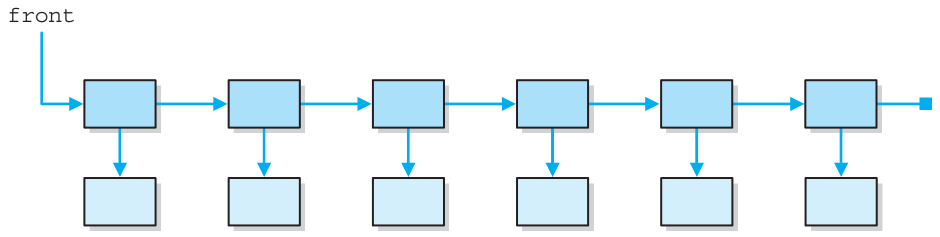


FIGURE 4.8 Using separate node objects to store and link elements

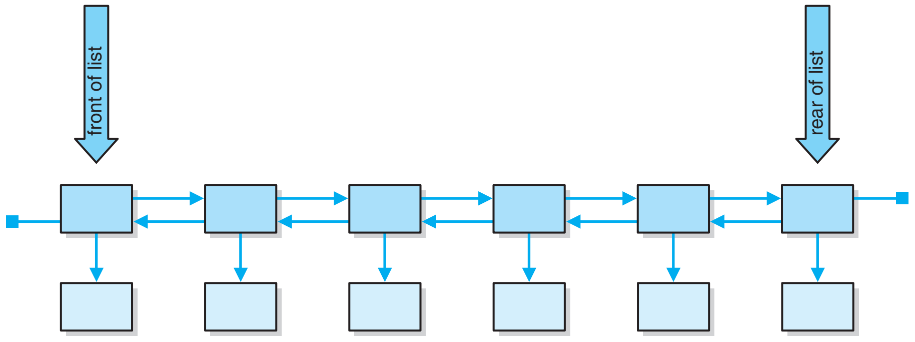


FIGURE 4.9 A doubly linked list

4.4 Implementing a Stack: With Links

Let's use a linked list to implement a stack collection, which was defined in Chapter 3. Note that we are not changing the way in which a stack works. Its conceptual nature remains the same, as does the set of operations defined for it. We are merely changing the underlying data structure used to implement it.

The purpose of the stack, and the solutions it helps us to create, also remains the same. The postfix expression evaluation example from Chapter 3 used the `ArrayStack<T>` class, but any valid implementation of a stack could be used instead. Once we create the `LinkedStack<T>` class to define an alternative implementation, it could be substituted into the postfix expression solution without having to change anything but the class name. That is the beauty of abstraction.

KEY CONCEPT
Any implementation of a collection can be used to solve a problem as long as it validly implements the appropriate operations.

In the following discussion, we show and discuss the methods that are important to understanding the linked list implementation of a stack. Some of the stack operations are left as programming projects.

The `LinkedStack` Class

The `LinkedStack<T>` class implements the `StackADT<T>` interface, just as the `ArrayStack<T>` class from Chapter 3 does. Both provide the operations defined for a stack collection.

Because we are using a linked-list approach, there is no array in which we store the elements of the collection. Instead, we need only a single reference to the first node in the list. We will also maintain a count of the number of elements in the list. The header and class-level data of the `LinkedStack<T>` class is therefore:

```
/**
 * @author Lewis and Chase
 *
 * Represents a linked implementation of a stack.
 */

package jss2;
import jss2.exceptions.*;
import java.util.Iterator;

public class LinkedStack<T> implements StackADT<T>
{
  /** indicates number of elements stored */
  private int count;
  /** pointer to top of stack */
  private LinearNode<T> top;
```

The `LinearNode<T>` class serves as the node class, containing a reference to the next `LinearNode<T>` in the list and a reference to the element stored in that node. Each node stores a generic type that is determined when the node is instantiated. In our `LinkedStack<T>` implementation, we simply use the same type for the node as used to define the stack. The `LinearNode<T>` class also contains methods to set and get the element values. The `LinearNode<T>` class is shown in Listing 4.1.

Note that the `LinearNode<T>` class is not tied to the implementation of a stack collection. It can be used in any linear linked-list implementation of a collection. We will use it for other collections as needed.

Using the `LinearNode<T>` class and maintaining a count of elements in the collection creates the implementation strategy depicted in Figure 4.10.

KEY CONCEPT
A linked implementation of a stack adds and removes elements from one end of the linked list.

The constructor of the `LinkedStack<T>` class sets the count of elements to zero and sets the front of the list, represented by the variable `top`, to `null`. Note that because a linked-list implementation does not have to worry about capacity limitations, there is no need to create a second constructor as we did in the `ArrayStack<T>` class of Chapter 3.

```
/**
 * Creates an empty stack.
 */
public LinkedStack()
{
   count = 0;
   top = null;
}
```

LISTING 4.1

```
  /**
  *  @author Lewis and Chase
  *
  *  Represents a node in a linked list.
  */

package jss2;

public class LinearNode<T>
{
   /** reference to next node in list */
   private LinearNode<T> next;

   /** element stored at this node */
   private T element;

   /**
    * Creates an empty node.
    */
   public LinearNode()
   {
     next = null;
     element = null;
   }

   /**
    * Creates a node storing the specified element.
    * @param elem element to be stored
    */
  public LinearNode (T elem)
  {
    next = null;
    element = elem;
  }

  /**
   * Returns the node that follows this one.
   * @return LinearNode<T> reference to next node
   */
  public LinearNode<T> getNext()
  {
    return next;
  }
```

LISTING 4.1 *continued*

```
  /**
   * Sets the node that follows this one.
   * @param node node to follow this one
   */
  public void setNext (LinearNode<T> node)
  {
    next = node;
  }

  /**
   * Returns the element stored in this node.
   * @return T element stored at this node
   */
  public T getElement()
  {
    return element;
  }

  /**
   * Sets the element stored in this node.
   * @param elem element to be stored at this node
   */
  public void setElement (T elem)
  {
    element = elem;
  }
}
```

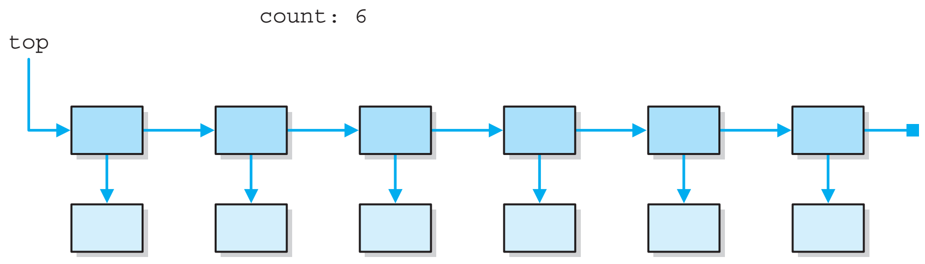


FIGURE 4.10 A linked implementation of a stack collection

Because the nature of a stack is to only allow elements to be added or removed from one end, we will only need to operate on one end of our linked list. We could choose to push the first element into the first position in the linked list, the second element into the second position, etc. This would mean that the top of the stack would always be at the tail end of the list. However, if we consider the efficiency of this strategy, we realize that this would mean that we would have to traverse the entire list on every push and every pop operation. Instead, we can choose to operate on the front of the list making the front of the list the top of the stack. In this way, we do not have to traverse the list for either the push or pop operations. Figure 4.11 illustrates this configuration for a stack containing four elements, A, B, C, and D, that have been pushed onto the stack in that order.

Let's explore the implementation of the stack operations for the `LinkedStack` class.

The `push` Operation

Every time a new element is pushed onto the stack, a new `LinearNode` object must be created to store it in the linked list. To position the newly created node at the top of the stack, we must set its `next` reference to the current top of the stack, and reset the `top` reference to point to the new node. We must also increment the `count` variable.

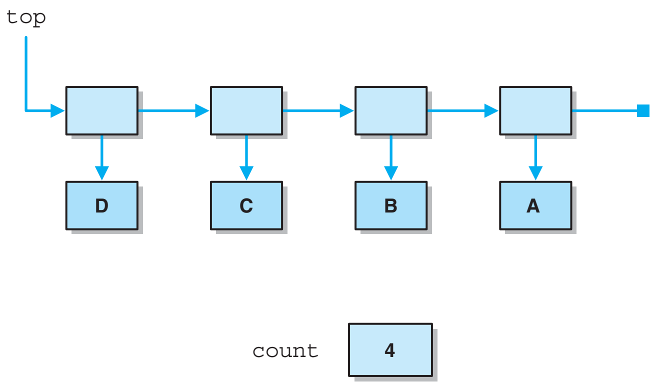


FIGURE 4.11 A linked implementation of a stack

Implementing these steps results in the following code:

```
/**
 * Adds the specified element to the top of this stack.
 * @param element element to be pushed on stack
 */
public void push (T element)
{
  LinearNode<T> temp = new LinearNode<T> (element);

  temp.setNext(top);
  top = temp;
  count++;
}
```

Figure 4.12 shows the result of pushing the element E onto the stack depicted in Figure 4.11.

The `push` operation for the linked implementation of a stack consists of the following steps:

- Create a new node containing a reference to the object to be placed on the stack.
- Set the `next` reference of the new node to point to the current top of the stack (which will be null if the stack is empty).
- Set the `top` reference to point to the new node.
- Increment the count of elements in the stack.

All of these steps have time complexity O(1) because they require only one processing step regardless of the number of elements already in the stack. Each of

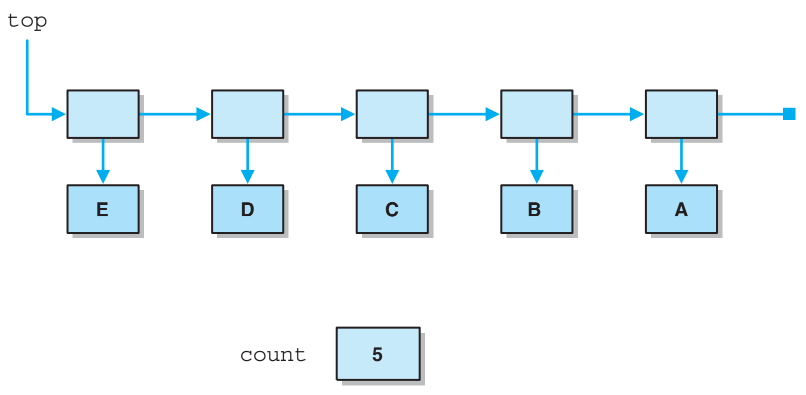


FIGURE 4.12 The stack after pushing element E

these steps would have to be accomplished once for each of the elements to be pushed. Thus, using this method, the push operation would be O(1).

The pop Operation

The pop operation is implemented by returning a reference to the element currently stored at the top of the stack and adjusting the top reference to the new top of the stack. Before attempting to return any element, however, we must first ensure that there is at least one element to return. This operation can be implemented as follows:

```
/**
 * Removes the element at the top of this stack and returns a
 * reference to it. Throws an EmptyCollectionException if the stack
 * is empty.
 * @return T element from top of stack
 * @throws EmptyStackException on pop from empty stack
 */
public T pop() throws EmptyCollectionException
{
  if (isEmpty())
    throw new EmptyCollectionException("Stack");

  T result = top.getElement();
  top = top.getNext();
  count--;

  return result;
}
```

If the stack is empty, as determined by the isEmpty method, an EmptyCollectionException is thrown. If there is at least one element to pop, it is stored in a temporary variable so that it can be returned. Then the reference to the top of the stack is set to the next element in the list, which is now the new top of the stack. The count of elements is decremented as well.

Figure 4.13 illustrates the result of a pop operation on the stack from Figure 4.12. Notice that this figure is identical to our original configuration in Figure 4.11. This illustrates the fact that the pop operation is the inverse of the push operation.

The pop operation for the linked implementation consists of the following steps:

- Make sure the stack is not empty.
- Set a temporary reference equal to the element on top of the stack.

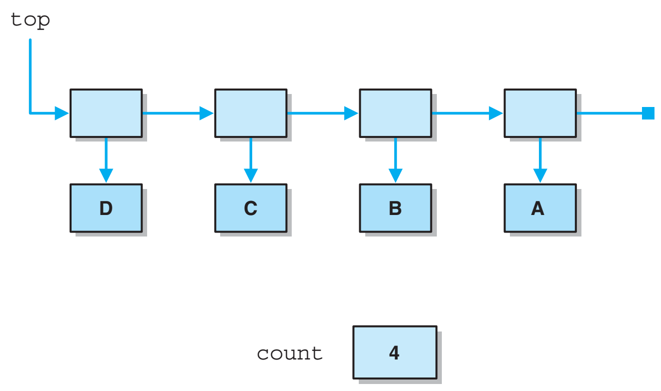


FIGURE 4.13 The stack after a pop operation

- Set the `top` reference equal to the `next` reference of the node at the top of the stack.
- Decrement the count of elements in the stack.
- Return the element pointed to by the temporary reference.

As with our previous examples, each of these operations consists of a single comparison or a simple assignment and is therefore O(1). Thus, the `pop` operation for the linked implementation is O(1).

Other Operations

Using a linked implementation, the `peek` operation is implemented by returning a reference to the element pointed to by the node pointed to by the top pointer. The `isEmpty` operation returns true if the count of elements is 0, and false otherwise. The `size` operation simply returns the count of elements in the stack. The `toString` operation can be implemented by simply traversing the linked list. These operations are left as programming projects.

4.5 Using Stacks: Traversing a Maze

Another classic use of a stack data structure is to keep track of alternatives in maze traversal or other similar algorithms that involve trial and error. Suppose that we build a grid as a two-dimensional array of ints where each number represents either a path (1) or a wall (0) in a maze:

```
private int [][] grid = { {1,1,1,0,1,1,0,0,0,1,1,1,1},
                          {1,0,0,1,1,0,1,1,1,1,0,0,1},
                          {1,1,1,1,1,0,1,0,1,0,1,0,0},
                          {0,0,0,0,1,1,1,0,1,0,1,1,1},
                          {1,1,1,0,1,1,1,0,1,0,1,1,1},
                          {1,0,1,0,0,0,0,1,1,1,0,0,1},
                          {1,0,1,1,1,1,1,1,0,1,1,1,1},
                          {1,0,0,0,0,0,0,0,0,0,0,0,0},
                          {1,1,1,1,1,1,1,1,1,1,1,1,1} };
```

Our goal is to start in the top-left corner of this grid and traverse to the bottom-right corner of this grid, traversing only positions that are marked as a path. Valid moves will be those that are within the bounds of the grid and are to cells in the grid marked with a 1. We will mark our path as we go by changing the 1's to 3's, and we will push only valid moves onto the stack.

Starting in the top-left corner, we have two valid moves: down and right. We push these moves onto the stack, pop the top move off of the stack (right), and then move to that location. This means that we moved right one position:

```
{3,3,1,0,1,1,0,0,0,1,1,1,1}
{1,0,0,1,1,0,1,1,1,1,0,0,1}
{1,1,1,1,1,0,1,0,1,0,1,0,0}
{0,0,0,0,1,1,1,0,1,0,1,1,1}
{1,1,1,0,1,1,1,0,1,0,1,1,1}
{1,0,1,0,0,0,0,1,1,1,0,0,1}
{1,0,1,1,1,1,1,1,0,1,1,1,1}
{1,0,0,0,0,0,0,0,0,0,0,0,0}
{1,1,1,1,1,1,1,1,1,1,1,1,1}
```

We now have only one valid move. We push that move onto the stack, pop the top element off of the stack (right), and then move to that location. Again we moved right one position:

```
{3,3,3,0,1,1,0,0,0,1,1,1,1}
{1,0,0,1,1,0,1,1,1,1,0,0,1}
{1,1,1,1,1,0,1,0,1,0,1,0,0}
{0,0,0,0,1,1,1,0,1,0,1,1,1}
{1,1,1,0,1,1,1,0,1,0,1,1,1}
{1,0,1,0,0,0,0,1,1,1,0,0,1}
{1,0,1,1,1,1,1,1,0,1,1,1,1}
{1,0,0,0,0,0,0,0,0,0,0,0,0}
{1,1,1,1,1,1,1,1,1,1,1,1,1}
```

From this position, we do not have any valid moves. At this point, however, our stack is not empty. Keep in mind that we still have a valid move on the stack left from the first position. We pop the next (and currently last) element off of the stack (down from the first position). We move to that position, push the valid move(s) from that position onto the stack, and continue processing.

Using a stack in this way is actually simulating *recursion*, a process whereby a method calls itself either directly or indirectly. Recursion, which we will discuss in greater detail in Chapter 7, uses the concept of a program stack. A *program stack* (or *run-time stack*) is used to keep track of methods that are invoked. Every time a method is called, an *activation record* that represents the invocation is created and pushed onto the program stack. Therefore, the elements on the stack represent the series of method invocations that occurred to reach a particular point in an executing program.

For example, when the `main` method of a program is called, an activation record for it is created and pushed onto the program stack. When `main` calls another method (say `m2`), an activation record for `m2` is created and pushed onto the stack. If `m2` calls method `m3`, then an activation record for `m3` is created and pushed onto the stack. When method `m3` terminates, its activation record is popped off of the stack and control returns to the calling method (`m2`), which is now on the top of the stack.

If an exception occurs during the execution of a Java program, the programmer can examine the *call stack trace* to see what method the problem occurred within and what method calls were made to arrive at that point.

KEY CONCEPT
Recursive processing can be simulated using a stack to keep track of the appropriate data.

An activation record contains various administrative data to help manage the execution of the program. It also contains a copy of the method's data (local variables and parameters) for that invocation of the method.

Because of the relationship between stacks and recursion, we can always rewrite a recursive program into a nonrecursive program that uses a stack. Instead of using recursion to keep track of the data, we can create our own stack to do so.

Listings 4.2 and 4.3 illustrate the `Maze` and `MazeSearch` classes that implement our stack-based solution to traversing a maze. We will revisit this same example in our discussion of recursion in Chapter 7.

This solution uses a class called `Position` to encapsulate the coordinates of a position within the maze. The `traverse` method loops, popping the top position off of the stack, marking it as tried, and then testing to see if we are done. If we are not done, then all of the valid moves from this position are pushed onto the stack and the loop continues. A private method called `pushNewPos` has been created to handle the task of putting the valid moves from the current position onto the stack:

```
private StackADT<Position> push_new_pos(int x, int y, StackADT<Position> stack)
  {
     Position npos = new Position();
     npos.setx(x);
     npos.sety(y);
     if (valid(npos.getx(),npos.gety()))
       stack.push(npos);
     return stack;
  }
```

LISTING 4.2

```
/**
 * @author Lewis and Chase
 *
 * Represents a maze of characters. The goal is to get from the
 * top left corner to the bottom right, following a path of 1's.
 */
import jss2.*;

public class Maze
{
  /**
   * constant to represent tried paths
   */
  private final int TRIED = 3;

  /**
   * constant to represent the final path
   */
  private final int PATH = 7;

  /**
   * two dimensional array representing the grid
   */
  private int [][] grid = { {1,1,1,0,1,1,0,0,0,1,1,1,1},
                            {1,0,0,1,1,0,1,1,1,1,0,0,1},
                            {1,1,1,1,1,0,1,0,1,0,1,0,0},
                            {0,0,0,0,1,1,1,0,1,0,1,1,1},
                            {1,1,1,0,1,1,1,0,1,0,1,1,1},
                            {1,0,1,0,0,0,0,1,1,1,0,0,1},
                            {1,0,1,1,1,1,1,1,0,1,1,1,1},
                            {1,0,0,0,0,0,0,0,0,0,0,0,0},
                            {1,1,1,1,1,1,1,1,1,1,1,1,1} };
```

LISTING 4.2 *continued*

```
/**
 * push a new attempted move onto the stack
 * @param x represents x coordinate
 * @param y represents y coordinate
 * @param stack the working stack of moves within the grid
 * @return StackADT<Position> stack of moves within the grid
 */
private StackADT<Position> push_new_pos(int x, int y,
                                        StackADT<Position> stack)
{
  Position npos = new Position();
  npos.setx(x);
  npos.sety(y);
  if (valid(npos.getx(),npos.gety()))
    stack.push(npos);

  return stack;
}

/**
 * Attempts to iteratively traverse the maze. It inserts special
 * characters indicating locations that have been tried and that
 * eventually become part of the solution. This method uses a
 * stack to keep track of the possible moves that could be made.
 * @return boolean returns true if the maze is successfully traversed
 */
public boolean traverse ()
{
  boolean done = false;
  Position pos = new Position();
  Object dispose;
  StackADT<Position> stack = new LinkedStack<Position> ();
  stack.push(pos);

  while (!(done))
  {
    pos = stack.pop();
    grid[pos.getx()][pos.gety()] = TRIED;  // this cell has been tried
    if (pos.getx() == grid.length-1 && pos.gety() == grid[0].length-1)
      done = true; // the maze is solved
    else
    {
      stack = push_new_pos(pos.getx(),pos.gety() - 1, stack);
      stack = push_new_pos(pos.getx(),pos.gety() + 1, stack);
```

LISTING 4.2 *continued*

```
        stack = push_new_pos(pos.getx() - 1,pos.gety(), stack);
        stack = push_new_pos(pos.getx() + 1,pos.gety(), stack);
      }
    }

    return done;
  }

  /**
   * Determines if a specific location is valid.
   * @param row int representing y coordinate
   * @param column int representing x coordinate
   * @return boolean true if the given coordinate is a valid move
   */
  private boolean valid (int row, int column)
  {
    boolean result = false;

    /** Check if cell is in the bounds of the matrix */
    if (row >= 0 && row < grid.length &&
        column >= 0 && column < grid[row].length)

      /** Check if cell is not blocked and not previously tried */
      if (grid[row][column] == 1)
        result = true;

    return result;
  }
  /**
   * Returns the maze as a string.
   * @return String representation of the maze grid
   */
  public String toString ()
  {
    String result = "\n";

    for (int row=0; row < grid.length; row++)
    {
      for (int column=0; column < grid[row].length; column++)
```

LISTING 4.2 *continued*

```
        result += grid[row][column] + "";

      result += "\n";
    }
    return result;
  }
}
```

LISTING 4.3

```
/**
 * @author Lewis and Chase
 *
 *  Demonstrates a simulation of recursion using a stack.
 */

public class MazeSearch
{

  /**
   * Creates a new maze, prints its original form, attempts to
   * solve it, and prints out its final form.
   * @param args array of Strings
   */
  public static void main (String[] args)
  {
    Maze labyrinth = new Maze();

    System.out.println (labyrinth);

    if (labyrinth.traverse ())
      System.out.println ("The maze was successfully traversed!");
    else
      System.out.println ("There is no possible path.");

    System.out.println (labyrinth);
   }
}
```

The UML description for the maze problem is left as an exercise.

4.6 Implementing Stacks: The `java.util.Stack` Class

The class `java.util.Stack` is an implementation of a stack provided in the Java Collections API framework. This implementation provides either the same or similar operations to the ones that we have been discussing:

- The `push` operation accepts a parameter item that is a reference to an object to be placed on the stack.
- The `pop` operation removes the object on top of the stack and returns a reference to it.
- The `peek` operation returns a reference to the object on top of the stack.
- The `empty` operation behaves the same as the `isEmpty` operation that we have been discussing.
- The `size` operation returns the number of elements in the stack.

The `java.util.Stack` class is derived from the `Vector` class and uses its inherited capabilities to store the elements in the stack. Because this implementation is built upon a vector, it exhibits the characteristics of both a vector and a stack. This implementation keeps track of the top of the stack using an index similar to the array implementation and thus does not require the additional overhead of storing a `next` reference in each node. Further, like the linked implementation, the `java.util.Stack` implementation allocates additional space only as needed.

Unique Operations

The `java.util.Stack` class provides an additional operation called `search`. Given an object to search for, the `search` operation returns the distance from the top of the stack to the first occurrence of that object on the stack. If the object is found at the top of the stack, the `search` method returns the value 1. If the object is not found on the stack, `search` returns the value -1.

KEY CONCEPT

The `java.util.Stack` class is derived from `Vector`, which gives a stack inappropriate operations.

Unfortunately, because the `java.util.Stack` implementation is derived from the `Vector` class, quite a number of other operations are inherited from the `Vector` class that are available for use. In some cases, these additional capabilities violate the basic assumptions of a stack. Most software engineers consider this a bad use of inheritance. Because a stack is not everything a vector is (conceptually), the `Stack` class should not be derived from the `Vector` class. Well-disciplined developers can, of course, limit themselves to only those operations appropriate to a stack.

Inheritance and Implementation

The class `java.util.Stack` is an extension of the class `java.util.Vector`, which is an extension of `java.util.AbstractList`, which is an extension of `java.util.AbstractCollection`, which is an extension of `java.lang.Object`. The `java.util.Stack` class implements the `cloneable`, `collection`, `list`, and `serializable` interfaces. These relationships are illustrated in the UML diagram of Figure 4.14.

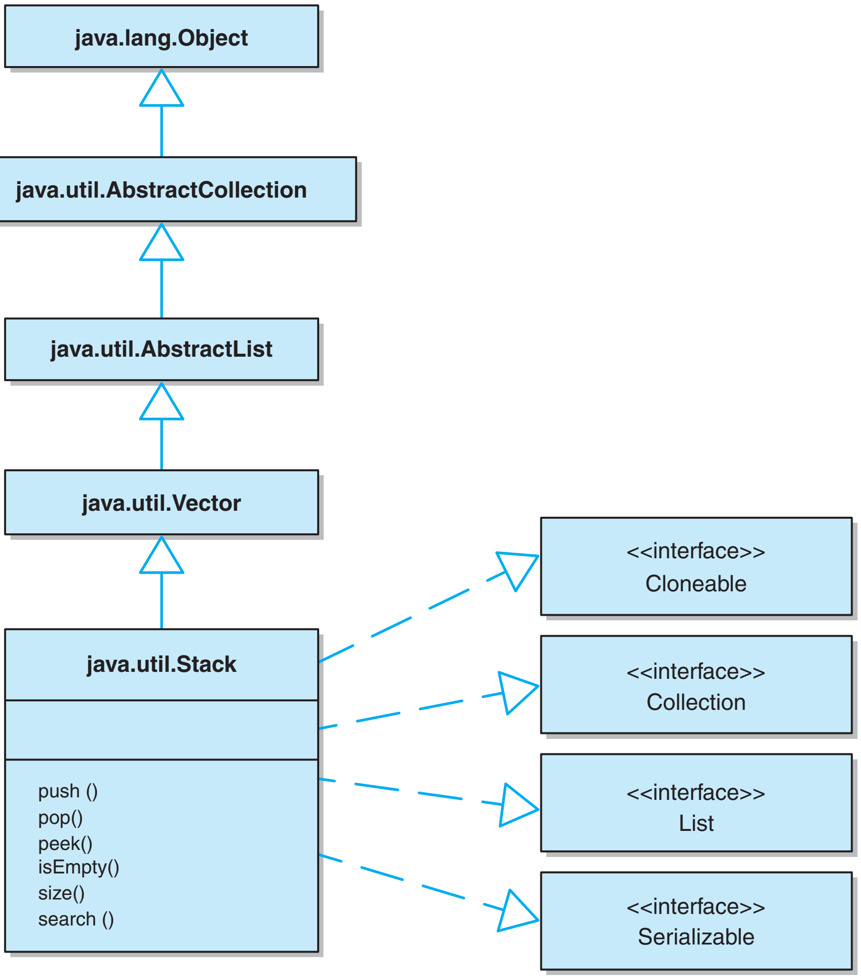


FIGURE 4.14 A UML description of the `java.util.Stack` class

Summary of Key Concepts

- Object reference variables can be used to create linked structures.
- A linked list is composed of objects that each point to the next object in the list.
- A linked list dynamically grows as needed and essentially has no capacity limitations.
- The order in which references are changed is crucial to maintaining a linked list.
- Dealing with the first node in a linked list often requires special handling.
- Implementing a list with a sentinel node or dummy node as the first node eliminates the special cases dealing with the first node.
- Objects that are stored in a collection should not contain any implementation details of the underlying data structure.
- Any implementation of a collection can be used to solve a problem as long as it validly implements the appropriate operations.
- A linked implementation of a stack adds and removes elements from one end of the linked list.
- Recursive processing can be simulated using a stack to keep track of the appropriate data.
- The `java.util.Stack` class is derived from `Vector`, which gives a stack inappropriate operations.

Self-Review Questions

SR 4.1 How do object references help us define data structures?

SR 4.2 Compare and contrast a linked list and an array.

SR 4.3 What special case exists when managing linked lists?

SR 4.4 Why should a linked list node be separate from the element stored on the list?

SR 4.5 What do the `LinkedStack<T>` and `ArrayStack<T>` classes have in common?

SR 4.6 What would be the time complexity of the `push` operation if we chose to push at the end of the list instead of the front?

SR 4.7 What is the difference between a doubly linked list and a singly linked list?

SR 4.8 What impact would the use of sentinel nodes or dummy nodes have upon a doubly linked list implementation?

SR 4.9 What are the advantages to using a linked implementation as opposed to an array implementation?

SR 4.10 What are the advantages to using an array implementation as opposed to a linked implementation?

SR 4.11 What are the advantages of the `java.util.Stack` implementation of a stack?

SR 4.12 What is the potential problem with the `java.util.Stack` implementation?

SR 4.13 What is the advantage of postfix notation?

Exercises

EX 4.1 Explain what will happen if the steps depicted in Figure 4.4 are reversed.

EX 4.2 Explain what will happen if the steps depicted in Figure 4.5 are reversed.

EX 4.3 Draw a UML diagram showing the relationships among the classes involved in the linked list implementation of a set.

EX 4.4 Write an algorithm for the `add` method that will add at the end of the list instead of the beginning. What is the time complexity of this algorithm?

EX 4.5 Modify the algorithm from the previous exercise so that it makes use of a rear reference. How does this affect the time complexity of this and the other operations?

EX 4.6 Discuss the effect on all the operations if there were not a count variable in the implementation.

EX 4.7 Discuss the impact (and draw an example) of using a sentinel node or dummy node at the head of the list.

EX 4.8 Draw the UML class diagram for the iterative maze solver example from this chapter.

Programming Projects

PP 4.1 Complete the implementation of the `LinkedStack<T>` class by providing the definitions for the `size`, `isEmpty`, and `toString` methods.

PP 4.2 Modify the `postfix` program from Chapter 3 so that it uses the `LinkedStack<T>` class instead of the `ArrayStack<T>` class.

PP 4.3 Create a new version of the `LinkedStack<T>` class that makes use of a dummy record at the head of the list.

PP 4.4 Create a simple graphical application that will allow a user to perform `push`, `pop`, and `peek` operations on a stack and display the resulting stack (using `toString`) in a text area.

PP 4.5 Design and implement an application that reads a sentence from the user and prints the sentence with the characters of each word backwards. Use a stack to reverse the characters of each word.

PP 4.6 Complete the solution to the iterative maze solver so that your solution marks the successful path.

PP 4.7 The linked implementation in this chapter uses a `count` variable to keep track of the number of elements in the stack. Rewrite the linked implementation without a `count` variable.

PP 4.8 There is a data structure called a drop-out stack that behaves like a stack in every respect except that if the stack size is n, when the n+1 element is pushed, the first element is lost. Implement a drop-out stack using links.

Answers to Self-Review Questions

SRA 4.1 An object reference can be used as a link from one object to another. A group of linked objects can form a data structure, such as a linked list, on which a collection can be based.

SRA 4.2 A linked list has no capacity limitations, while an array does. However, arrays provide direct access to elements using indexes, whereas a linked list must be traversed one element at a time to reach a particular point in the list.

SRA 4.3 The primary special case in linked list processing occurs when dealing with the first element in the list. A special reference variable is maintained that specifies the first element in the list. If that element is deleted, or a new element is added in front of it, the `front` reference must be carefully maintained.

SRA 4.4 It is unreasonable to assume that every object that we may want to put in a collection can be designed to cooperate with the collection implementation. Furthermore, the implementation details are supposed to be kept distinct from the user of the collection, including the elements the user chooses to add to the collection.

SRA 4.5 Both the `LinkedStack<T>` and `ArrayStack<T>` classes implement the `StackADT<T>` interface. This means that they both represent a stack collection, providing the necessary operations needed to use a stack. Though they both have distinct approaches to managing the collection, they are functionally interchangeable from the user's point of view.

SRA 4.6 To push at the end of the list, we would have to traverse the list to reach the last element. This traversal would cause the time complexity to be O(n). An alternative would be to modify the solution to add a `rear` reference that always pointed to the last element in the list. This would help the time complexity for `add` but would have consequences if we try to remove the last element.

SRA 4.7 A singly linked list maintains a reference to the first element in the list and then a `next` reference from each node to the following node in the list. A doubly linked list maintains two references: `front` and `rear`. Each node in the doubly linked list stores both a `next` and a `previous` reference.

SRA 4.8 It would take two dummy records in a doubly linked list, one at the front and one at the rear, to eliminate the special cases when dealing with the first and last node.

SRA 4.9 A linked implementation allocates space only as it is needed and has a theoretical limit of the size of the hardware.

SRA 4.10 An array implementation uses less space per object since it only has to store the object and not an extra pointer. However, the array implementation will allocate much more space than it needs initially.

SRA 4.11 Because the `java.util.Stack` implementation is an extension of the `Vector` class, it can keep track of the positions of elements in the stack using an index and thus does not require each node to store an additional pointer. This implementation also allocates space only as it is needed, like the linked implementation.

SRA 4.12 The `java.util.Stack` implementation is an extension of the `Vector` class and thus inherits a large number of operations that violate the basic assumptions of a stack.

SRA 4.13 Postfix notation avoids the need for precedence rules that are required to evaluate infix expressions.

5 Queues

A queue is another collection with which we are inherently familiar. A queue is a waiting line, such as a line of customers waiting in a bank for their opportunity to talk to a teller. In fact, in many countries the word queue is used habitually in this way. In such countries, a person might say "join the queue" rather than "get in line." Other examples of queues include a checkout line at the grocery store or cars waiting at a stoplight. In any queue, an item enters on one end and leaves from the other. Queues have a variety of uses in computer algorithms.

CHAPTER OBJECTIVES

- Examine queue processing
- Define a queue abstract data type
- Demonstrate how a queue can be used to solve problems
- Examine various queue implementations
- Compare queue implementations

5.1 A Queue ADT

A *queue* is a linear collection whose elements are added on one end and removed from the other. Therefore, we say that queue elements are processed in a *first in, first out* (FIFO) manner. Elements are removed from a queue in the same order in which they are placed on the queue.

KEY CONCEPT

Queue elements are processed in a FIFO manner—the first element in is the first element out.

This is consistent with the general concept of a waiting line. When a customer arrives at a bank, he or she begins waiting at the end of the line. When a teller becomes available, the customer at the beginning of the line leaves the line to receive service. Eventually every customer that started out at the end of the line moves to the front of the line and exits. For any given set of people, the first person to get in line is the first person to leave it.

The processing of a queue is pictured in Figure 5.1. Usually a queue is depicted horizontally. One end is established as the *front* of the queue and the other as the *rear* of the queue. Elements go onto the rear of the queue and come off of the front. Sometimes the front of the queue is called the *head* and the rear of the queue the *tail*.

Compare and contrast the processing of a queue to the LIFO (last in, first out) processing of a stack, which was discussed in Chapters 3 and 4. In a stack, the processing occurs at only one end of the collection. In a queue, processing occurs at both ends.

The operations defined for a queue ADT are listed in Figure 5.2. The term *enqueue* is used to refer to the process of adding a new element to the end of a queue. Likewise, *dequeue* refers to removing the element at the front of a queue. The *first* operation allows the user to examine the element at the front of the queue without removing it from the collection.

Remember that naming conventions are not universal for collection operations. Sometimes `enqueue` is simply called `add` or `insert`. The `dequeue` operation is

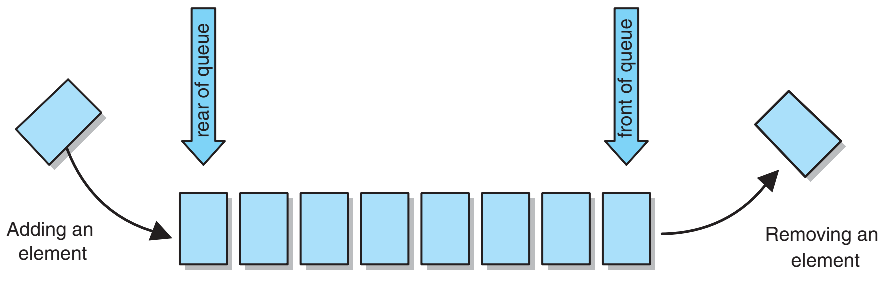


FIGURE 5.1 A conceptual view of a queue

Operation	Description
`enqueue`	Adds an element to the rear of the queue.
`dequeue`	Removes an element from the front of the queue.
`first`	Examines the element at the front of the queue.
`isEmpty`	Determines if the queue is empty.
`size`	Determines the number of elements on the queue.
`toString`	Returns a string representation of the queue.

FIGURE 5.2 The operations on a queue

sometimes called `remove` or `serve`. The `first` operation is sometimes called `front`.

Note that there is a general similarity between the operations of a queue and those of a stack. The `enqueue`, `dequeue`, and `first` operations correspond to the stack operations `push`, `pop`, and `peek`. Similar to a stack, there are no operations that allow the user to "reach into" the middle of a queue and reorganize or remove elements. If that type of processing is required, perhaps the appropriate collection to use is a list of some kind, such as those discussed in the next chapter.

As we did with stacks, we define a generic `QueueADT` interface that represents the queue operations, separating the general purpose of the operations from the variety of ways they could be implemented. A Java version of the `QueueADT` interface is shown in Listing 5.1, and its UML description is shown in Figure 5.3.

Note that in addition to the standard queue operations, we have also included a `toString` method, as we did with our stack collection. It is included for convenience and is not generally considered a classic operation on a queue.

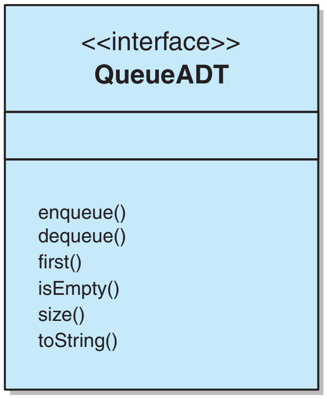


FIGURE 5.3 The `QueueADT` interface in UML

LISTING 5.1

```
/**
 * QueueADT defines the interface to a queue collection.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 8/12/08
 */

package jss2;

public interface QueueADT<T>
{
   /**
    * Adds one element to the rear of this queue.
    *
    * @param element the element to be added to the rear of this queue
    */
   public void enqueue (T element);

   /**
    * Removes and returns the element at the front of this queue.
    *
    * @return the element at the front of this queue
    */
   public T dequeue();

   /**
    * Returns without removing the element at the front of this queue.
    *
    * @return the first element in this queue
    */
   public T first();

   /**
    * Returns true if this queue contains no elements.
    *
    * @return true if this queue is empty
    */
   public boolean isEmpty();

   /**
    * Returns the number of elements in this queue.
    *
    * @return the integer representation of the size of this queue
    */
```

LISTING 5.1 *continued*

```
    public int size();

    /**
     * Returns a string representation of this queue.
     *
     * @return the string representation of this queue
     */
    public String toString();
}
```

Queues have a wide variety of application within computing. Whereas the principle purpose of a stack is to reverse order, the principle purpose of a queue is to preserve order. Before exploring various ways to implement a queue, let's examine some ways in which a queue can be used to solve problems.

5.2 Using Queues: Code Keys

A *Caesar cipher* is a simple approach to encoding messages by shifting each letter in a message along the alphabet by a constant amount k. For example, if k equals 3, then in an encoded message, each letter is shifted three characters forward: a is replaced with d, b with e, c with f, and so on. The end of the alphabet wraps back around to the beginning. Thus, w is replaced with z, x with a, y with b, and z with c.

To decode the message, each letter is shifted the same number of characters backwards. Therefore, if k equals 3, the encoded message

```
vlpsolflwb iroorzv frpsohalwb
```

would be decoded into

```
simplicity follows complexity
```

Julius Caesar actually used this type of cipher in some of his secret government correspondence (hence the name). Unfortunately, the Caesar cipher is fairly easy to break. There are only 26 possibilities for shifting the characters, and the code can be broken by trying various key values until one works.

An improvement can be made to this encoding technique if we use a *repeating key*. Instead of shifting each character by a constant amount, we can shift each character by a different amount using a list of key values. If the message is longer

Encoded Message:	n	o	v	a	n	j	g	h	l		m	u		u	r	x	l	v
Key:	3	1	7	4	2	5	3	1	7		4	2		5	3	1	7	4
Decoded Message:	k	n	o	w	l	e	d	g	e		i	s		p	o	w	e	r

FIGURE 5.4 An encoded message using a repeating key

than the list of key values, we just start using the key over again from the beginning. For example, if the key values are

3 1 7 4 2 5

then the first character is shifted by three, the second character by one, the third character by seven, etc. After shifting the sixth character by five, we start using the key over again. The seventh character is shifted by three, the eighth by one, etc.

Figure 5.4 shows the message "knowledge is power" encoded using this repeating key. Note that this encryption approach encodes the same letter into different characters, depending on where it occurs in the message and thus which key value is used to encode it. Conversely, the same character in the encoded message is decoded into different characters.

KEY CONCEPT

A queue is a convenient collection for storing a repeating code key.

The program in Listing 5.2 uses a repeating key to encode and decode a message. The key of integer values is stored in a queue. After a key value is used, it is put back on the end of the queue so that the

LISTING 5.2

```
/**
 * Codes demonstrates the use of queues to encrypt and decrypt messages.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 08/12/08
 */

import jss2.CircularArrayQueue;

public class Codes
{
   /**
    * Encode and decode a message using a key of values stored in
    * a queue.
    */
```

LISTING 5.2 *continued*

```
   public static void main (String[] args)
   {
     int[] key = {5, 12, -3, 8, -9, 4, 10};
     Integer keyValue;
     String encoded = "", decoded = "";
     String message = "All programmers are playwrights and all " +
                      "computers are lousy actors.";
     CircularArrayQueue<Integer> keyQueue1 = new CircularArrayQueue<Integer>();
     CircularArrayQueue<Integer> keyQueue2 = new CircularArrayQueue<Integer>();

     /** load key queue */

     for (int scan=0; scan < key.length; scan++)
     {
       keyQueue1.enqueue (new Integer(key[scan]));
       keyQueue2.enqueue (new Integer(key[scan]));
     }

     // encode message
     for (int scan=0; scan < message.length(); scan++)
     {
       keyValue = keyQueue1.dequeue();
       encoded += (char) ((int)message.charAt(scan) + keyValue.intValue());
       keyQueue1.enqueue (keyValue);
     }

     System.out.println ("Encoded Message:\n" + encoded + "\n");

     // decode message
     for (int scan=0; scan < encoded.length(); scan++)
     {
       keyValue = keyQueue2.dequeue();
       decoded += (char) ((int)encoded.charAt(scan) - keyValue.intValue());
       keyQueue2.enqueue (keyValue);
     }

     System.out.println ("Decoded Message:\n" + decoded);
   }
}
```

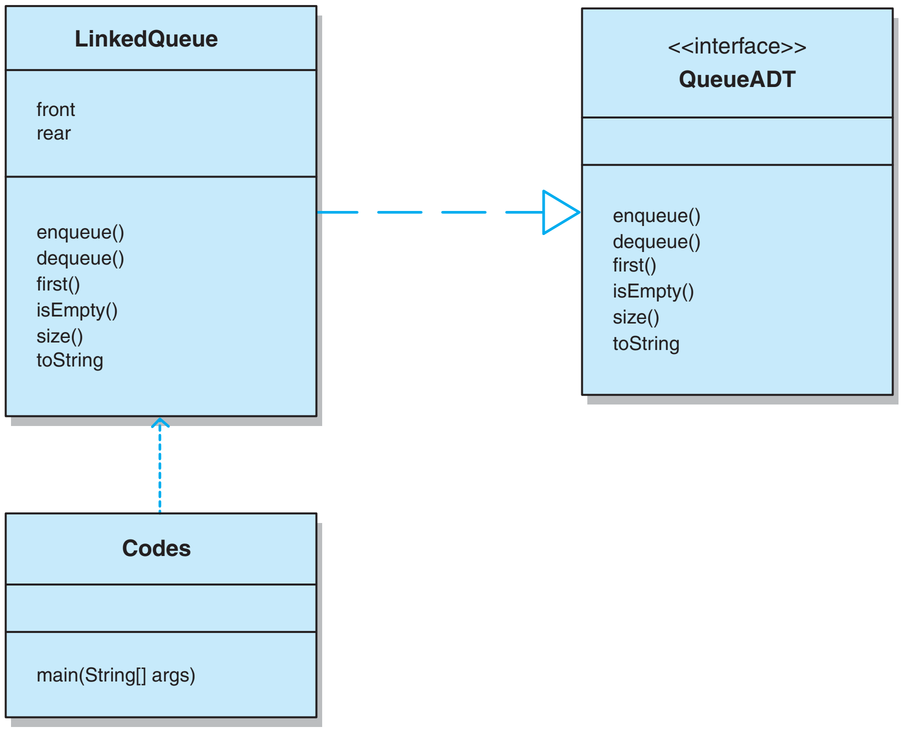


FIGURE 5.5 UML description of the `Codes` program

key continually repeats as needed for long messages. The key in this example uses both positive and negative values. Figure 5.5 illustrates the UML description of the `Codes` class.

This program actually uses two copies of the key stored in two separate queues. The idea is that the person encoding the message has one copy of the key, and the person decoding the message has another. Two copies of the key are helpful in this program as well because the decoding process needs to match up the first character of the message with the first value in the key.

Also, note that this program doesn't bother to wrap around the end of the alphabet. It encodes any character in the Unicode character set by shifting it to some other position in the character set. Therefore, we can encode any character, including uppercase letters, lowercase letters, and punctuation. Even spaces get encoded.

Using a queue to store the key makes it easy to repeat the key by putting each key value back onto the queue as soon as it is used. The nature of a queue keeps the key values in the proper order, and we don't have to worry about reaching the end of the key and starting over.

5.3 Using Queues: Ticket Counter Simulation

Let's look at another example using queues. Consider the situation in which you are waiting in line to purchase tickets at a movie theatre. In general, the more cashiers there are, the faster the line moves. The theatre manager wants to keep his customers happy, but he doesn't want to employ any more cashiers than necessary. Suppose the manager wants to keep the total time needed by a customer to less than seven minutes. Being able to simulate the effect of adding more cashiers during peak business hours allows the manager to plan more effectively. And, as we've discussed, a queue is the perfect collection for representing a waiting line.

KEY CONCEPT

Simulations are often implemented using queues to represent waiting lines.

Our simulated ticket counter will use the following assumptions:

- There is only one line and it is first come first served (a queue).
- Customers arrive on average every 15 seconds.
- If there is a cashier available, processing begins immediately upon arrival.
- Processing a customer request takes on average two minutes (120 seconds) from the time the customer reaches a cashier.

First we can create a `Customer` class, as shown in Listing 5.3. A `Customer` object keeps track of the time the customer arrives and the time the customer departs after purchasing a ticket. The total time spent by the customer is therefore the departure time minus the arrival time. To keep things simple, our simulation

LISTING 5.3

```
/**
 * Customer represents a waiting customer.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 08/12/08
 */

public class Customer
{
  private int arrivalTime, departureTime;

  /**
   * Creates a new customer with the specified arrival time.
   *
   * @param arrives the integer representation of the arrival time
   */
```

LISTING 5.3 *continued*

```
public Customer (int arrives)
{
   arrivalTime = arrives;
   departureTime = 0;
}

/**
 * Returns the arrival time of this customer.
 *
 * @return the integer representation of the arrival time
 */

public int getArrivalTime()
{
   return arrivalTime;
}

/**
 * Sets the departure time for this customer.
 *
 * @param departs the integer representation of the departure time
 */

public void setDepartureTime (int departs)
{
   departureTime = departs;
}

/**
 * Returns the departure time of this customer.
 *
 * @return the integer representation of the departure time
 */

public int getDepartureTime()
{
   return departureTime;
}

/**
 * Computes and returns the total time spent by this customer.
 *
 * @return the integer representation of the total customer time
 */
```

LISTING 5.3 *continued*

```
   public int totalTime()
   {
      return departureTime - arrivalTime;
   }
}
```

will measure time in elapsed seconds, so a time value can be stored as a single integer. Our simulation will begin at time 0.

Our simulation will create a queue of customers, then see how long it takes to process those customers if there is only one cashier. Then we will process the same queue of customers with two cashiers. Then we will do it again with three cashiers. We continue this process for up to ten cashiers. At the end we compare the average time it takes to process a customer.

Because of our assumption that customers arrive every 15 seconds (on average), we can preload a queue with customers. We will process 100 customers in this simulation.

The program shown in Listing 5.4 conducts our simulation. The outer loop determines how many cashiers are used in each pass of the simulation. For each pass, the customers are taken from the queue in turn and processed by a cashier. The total elapsed time is tracked, and at the end of each pass the average time is computed. Figure 5.6 shows the UML description of the `TicketCounter` and `Customer` classes.

LISTING 5.4

```
/**
 * TicketCounter demonstrates the use of a queue for simulating a waiting line.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 08/12/08
 */

import jss2.*;

public class TicketCounter
{
```

LISTING 5.4 *continued*

```
final static int PROCESS = 120;
final static int MAX_CASHIERS = 10;
final static int NUM_CUSTOMERS = 100;

public static void main ( String[] args)
{
   Customer customer;
   LinkedQueue<Customer> customerQueue = new LinkedQueue<Customer>();
   int[] cashierTime = new int[MAX_CASHIERS];
   int totalTime, averageTime, departs;

   // process the simulation for various number of cashiers

   for (int cashiers=0; cashiers < MAX_CASHIERS; cashiers++)
   {
      // set each cashiers time to zero initially
      for (int count=0; count < cashiers; count++)
         cashierTime[count] = 0;

      // load customer queue

      for (int count=1; count <= NUM_CUSTOMERS; count++)
         customerQueue.enqueue(new Customer(count*15));

      totalTime = 0;

      // process all customers in the queue

      while (!(customerQueue.isEmpty()))
      {
         for (int count=0; count <= cashiers; count++)
         {
            if (!(customerQueue.isEmpty()))
            {
               customer = customerQueue.dequeue();
               if (customer.getArrivalTime() > cashierTime[count])
                  departs = customer.getArrivalTime() + PROCESS;
               else
                  departs = cashierTime[count] + PROCESS;
               customer.setDepartureTime (departs);
               cashierTime[count] = departs;
               totalTime += customer.totalTime();
```

LISTING 5.4 *continued*

```
                }
            }
        }

        // output results for this simulation

        averageTime = totalTime / NUM_CUSTOMERS;
        System.out.println ("Number of cashiers: " + (cashiers+1));
        System.out.println ("Average time: " + averageTime + "\n");
    }
  }
}
```

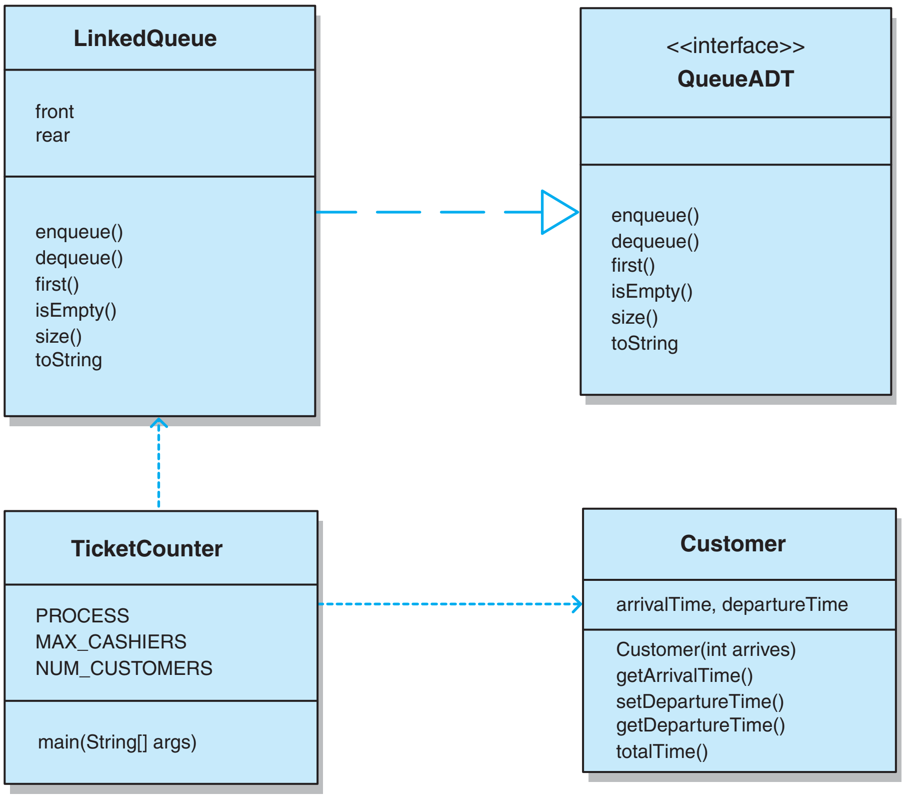


FIGURE 5.6 UML description of the `TicketCounter` program

Number of Cashiers:	1	2	3	4	5	6	7	8	9	10
Average Time (sec):	5317	2325	1332	840	547	355	219	120	120	120

FIGURE 5.7 The results of the ticket counter simulation

The results of the simulation are shown in Figure 5.7. Note that with eight cashiers, the customers do not wait at all. The time of 120 seconds reflects only the time it takes to walk up and purchase the ticket. Increasing the number of cashiers to nine or ten or more will not improve the situation. Since the manager has decided he wants to keep the total average time to less than seven minutes (420 seconds), the simulation tells him that he should have six cashiers.

5.4 Implementing Queues: With Links

Because a queue is a linear collection, we can implement a queue as a linked list of `LinearNode` objects, as we did with stacks. The primary difference is that we will have to operate on both ends of the list. Therefore, in addition to a reference (called `front`) pointing to the first element in the list, we will also keep track of a second reference (called `rear`) that points to the last element in the list. We will also use an integer variable called `count` to keep track of the number of elements in the queue.

KEY CONCEPT
A linked implementation of a queue is facilitated by references to the first and last elements of the linked list.

Does it make a difference to which end of the list we add or enqueue elements and from which end of the list we remove or dequeue elements? If our linked list is singly linked, meaning that each node has only a pointer to the node behind it in the list, then yes, it does make a difference. In the case of the `enqueue` operation, it will not matter whether we add new elements to the front or the rear of the list. The processing steps will be very similar. If we add to the front of the list then we would set the `next` pointer of the new node to point to the `front` of the list and set the `front` variable to point to the new node. If we add to the `rear` of the list then we would set the `next` pointer of the node at the `rear` of the list to point to the new node and then set the `rear` of the list to point to the new node. In both cases, all of these processing steps are O(1), and therefore the time complexity of the `enqueue` operation would be O(1).

The difference between our two choices, adding to the `front` or the `rear` of the list, occurs with the `dequeue` operation. If we `enqueue` at the `rear` of the list and `dequeue` from the `front` of the list, then to `dequeue` we simply set a temporary

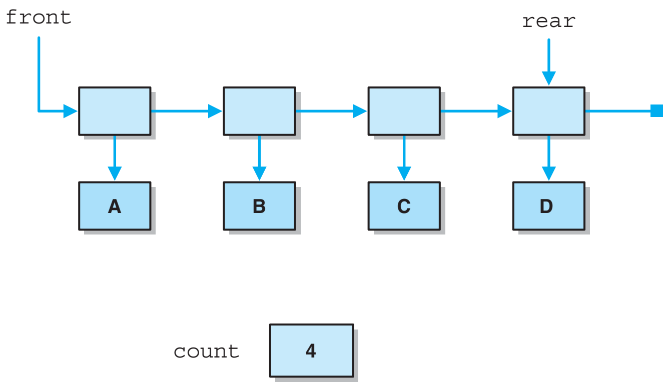


FIGURE 5.8 A linked implementation of a queue

variable to point to the element at the `front` of the list and then set the `front` variable to the value of the next pointer of the first node. Both processing steps are O(1) and therefore the operation would be O(1). However, if we `enqueue` at the `front` of the list and therefore `dequeue` at the `rear` of the list, our processing steps become more interesting. In order to `dequeue` from the `rear` of the list we must set a temporary variable to point to the element at the `rear` of the list and then set the `rear` pointer to point to the node before the current `rear`. Unfortunately, in a singly linked list, we cannot get to this node without traversing the list. Therefore if we chose to `enqueue` at the `front` and `dequeue` at the `rear`, the `dequeue` operation would be O(n) instead of O(1) as it is with our other choice. Thus, we choose to `enqueue` at the `rear` and `dequeue` at the `front` of our singly linked list. Keep in mind that a doubly linked list would solve the problem of having to traverse the list and thus it would not matter which end was which in a doubly linked implementation.

Figure 5.8 depicts this strategy for implementing a queue. It shows a queue that has had the elements A, B, C, and D added to the queue or enqueued, in that order.

Remember that Figure 5.8 depicts the general case. We always have to be careful to accurately maintain our references in special cases. For an empty queue, the `front` and `rear` references are both null and the `count` is zero. If there is exactly one element in the queue, both the `front` and `rear` references point to the same object. If we were using a singly linked implementation with a sentinel node and the queue was empty, then both `front` and `rear` would point to the sentinel node.

Let's explore the implementation of the queue operations using this linked list approach. The header, class-level data, and constructors for our linked implementation of a queue are provided for context:

```
/**
 * LinkedQueue represents a linked implementation of a queue.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 08/12/08
 */

package jss2;
import jss2.exceptions.*;

public class LinkedQueue<T> implements QueueADT<T>
{
   private int count;
   private LinearNode<T> front, rear;

   /**
    * Creates an empty queue.
    */

   public LinkedQueue()
   {
      count = 0;
      front = rear = null;
   }
```

The enqueue Operation

The `enqueue` operation requires that we put the new element on the rear of the list. In the general case, that means setting the `next` reference of the current last element to the new one, and resetting the `rear` reference to the new last element. However, if the queue is currently empty, the `front` reference must also be set to the new (and only) element. This operation can be implemented as follows:

```
/**
 * Adds the specified element to the rear of this queue.
 *
 * @param element the element to be added to the rear of this queue
 */

public void enqueue (T element)
{
      LinearNode<T> node = new LinearNode<T> (element);

      if (isEmpty())
         front = node;
      else
         rear.setNext (node);

      rear = node;
      count++;
   }
```

Note that the next reference of the new node need not be explicitly set in this method because it has already been set to null in the constructor for the `LinearNode` class. The `rear` reference is set to the new node in either case, and the `count` is incremented. Implementing the queue operations with sentinel nodes is left as an exercise. As we discussed earlier, this operation is O(1).

Figure 5.9 shows the queue from Figure 5.8 after element E has been added.

The dequeue Operation

The first issue to address when implementing the `dequeue` operation is to ensure that there is at least one element to return. If not, an `EmptyCollectionException` is thrown. As we did with our stack collection in Chapters 3 and 4, it makes sense to employ a generic `EmptyCollectionException` to which we can pass a parameter

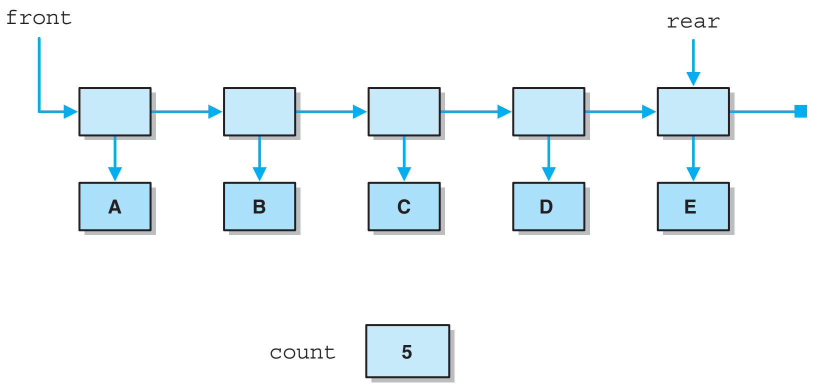


FIGURE 5.9 The queue after adding element E

specifying which collection we are dealing with. If there is at least one element in the queue, the first one in the list is returned and the front reference is updated:

```
/**
 * Removes the element at the front of this queue and returns a
 * reference to it. Throws an EmptyCollectionException if the
 * queue is empty.
 *
 * @return the element at the front of this queue
 * @throws EmptyCollectionException if an empty collection exception occurs
 */

public T dequeue() throws EmptyCollectionException
{
   if (isEmpty())
      throw new EmptyCollectionException ("queue");

   T result = front.getElement();
   front = front.getNext();
   count--;

   if (isEmpty())
      rear = null;

   return result;
}
```

KEY CONCEPT

The enqueue and dequeue operations work on opposite ends of the collection.

For the dequeue operation, we must consider the situation in which we are returning the only element in the queue. If, after removing the front element, the queue is now empty, then the rear reference is set to null. Note that in this case, the front will be null

DESIGN FOCUS

The same goals of reuse apply to exceptions as they do to other classes. The EmptyCollectionException class is a good example of this. It is an example of an exceptional case that will be the same for any collection that we create (e.g., attempting to perform an operation on the collection that cannot be performed if the collection is empty). Thus, creating a single exception with a parameter that allows us to designate which collection has thrown the exception is an excellent example of designing for reuse.

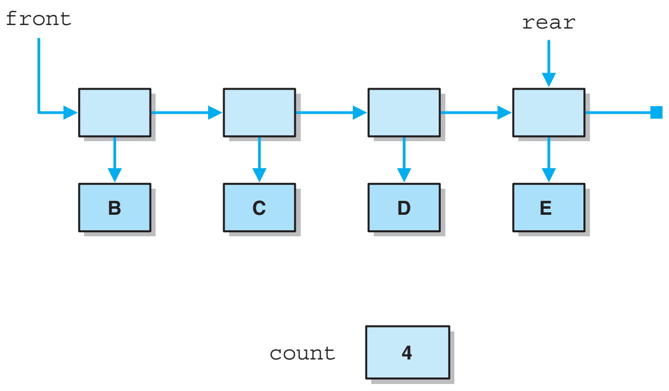


FIGURE 5.10 The queue after a `dequeue` operation

because it was set equal to the `next` reference of the last element in the list. Again, as we discussed earlier, the `dequeue` operation for our implementation is O(1).

Figure 5.10 shows the result of a `dequeue` operation on the queue from Figure 5.9. The element A at the front of the list is removed and returned to the user.

Note that, unlike the `pop` and `push` operations on a stack, the `dequeue` operation is not the inverse of `enqueue`. That is, Figure 5.10 is not identical to our original configuration shown in Figure 5.8, because the `enqueue` and `dequeue` operations are working on opposite ends of the collection.

Other Operations

The remaining operations in the linked queue implementation are fairly straightforward and are similar to those in the stack collection. The `first` operation is implemented by returning a reference to the element at the front of the queue. The `isEmpty` operation returns true if the count of elements is 0, and false otherwise. The `size` operation simply returns the count of elements in the queue. Finally, the `toString` operation returns a string made up of the `toString` results of each individual element. These operations are left as programming projects.

5.5 Implementing Queues: With Arrays

One array-based strategy for implementing a queue is to fix one end of the queue (say, the front) at index 0 of the array. The elements are stored contiguously in the array. Figure 5.11 depicts a queue stored in this manner, assuming elements A, B, C, and D have been added to the queue in that order.

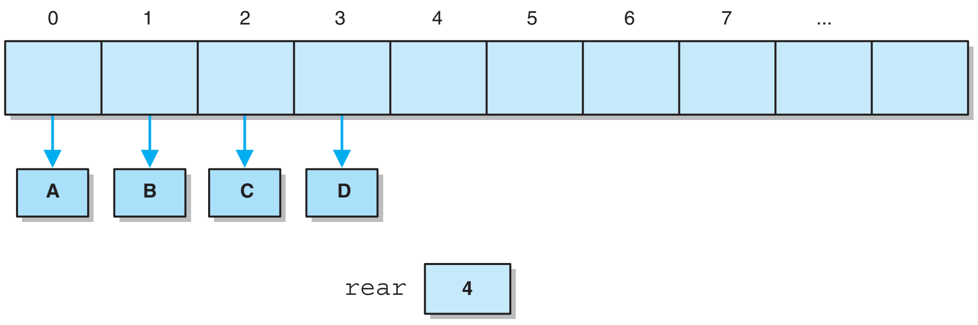


FIGURE 5.11 An array implementation of a queue

> **KEY CONCEPT**
> Because queue operations modify both ends of the collection, fixing one end at index 0 requires that elements be shifted.

The integer variable `rear` is used to indicate the next open cell in the array. Note that it also represents the number of elements in the queue.

This strategy assumes that the first element in the queue is always stored at index 0 of the array. Because queue processing affects both ends of the collection, this strategy will require that we shift the elements whenever an element is removed from the queue. This required shifting of elements would make the `dequeue` operation O(n). Like our discussion of the complexity of our singly linked list implementation above, making a poor choice in our array implementation could lead to less than optimal efficiency.

> **KEY CONCEPT**
> The shifting of elements in a noncircular array implementation creates an O(n) complexity.

Would it make a difference if we fixed the rear of the queue at index 0 of the array instead of the front? Keep in mind that when we `enqueue` an element onto the queue, we do so at the rear of the queue. This would mean that each `enqueue` operation would result in shifting all of the elements in the queue up one position in the array making the `enqueue` operation O(n).

DESIGN FOCUS

It is important to note that this fixed array implementation strategy, which was very effective in our implementation of a stack, is not nearly as efficient for a queue. This is an important example of matching the data structure used to implement a collection with the collection itself. The fixed array strategy was efficient for a stack because all of the activity (adding and removing elements) was on one end of the collection and thus on one end of the array. With a queue, now that we are operating on both ends of the collection and order does matter, the fixed array implementation is much less efficient.

The key is to not fix either end. As elements are dequeued, the front of the queue will move further into the array. As elements are enqueued, the rear of the queue will also move further into the array. The challenge comes when the rear of the queue reaches the end of the array. Enlarging the array at this point is not a practical solution, and does not make use of the now empty space in the lower indexes of the array.

KEY CONCEPT

Treating arrays as circular eliminates the need to shift elements in an array queue implementation.

To make this solution work, we will use a *circular array* to implement the queue, defined in a class called `CircularArrayQueue`. A circular array is not a new construct—it is just a way to think about the array used to store the collection. Conceptually, the array is used as a circle, whose last index is followed by the first index. A circular array storing a queue is shown in Figure 5.12.

Two integer values are used to represent the front and rear of the queue. These values change as elements are added and removed. Note that the value of `front` represents the location where the first element in the queue is stored, and the value of `rear` represents the next available slot in the array (not where the last element is stored). Using `rear` in this manner is consistent with our other array implementation. Note, however, that the value of `rear` no longer represents the

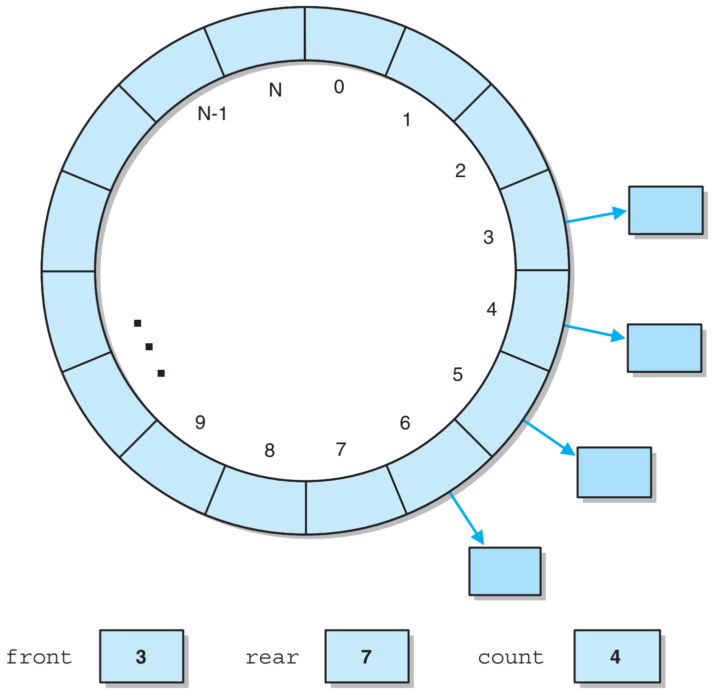


FIGURE 5.12 A circular array implementation of a queue

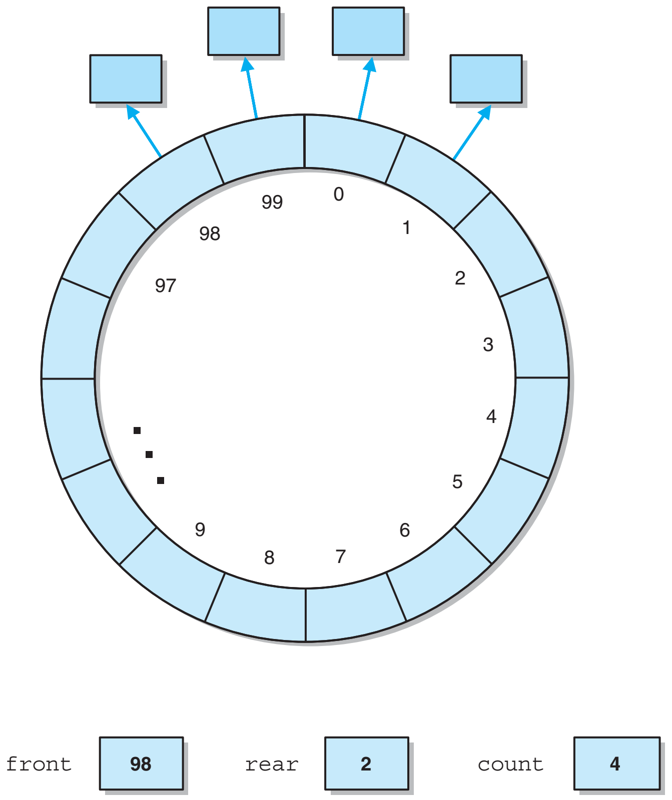


FIGURE 5.13 A queue straddling the end of a circular array

number of elements in the queue. We will use a separate integer value to keep a count of the elements.

When the rear of the queue reaches the end of the array, it "wraps around" to the front of the array. The elements of the queue can therefore straddle the end of the array, as shown in Figure 5.13, which assumes the array can store 100 elements.

Using this strategy, once an element has been added to the queue, it stays in one location in the array until it is removed with a `dequeue` operation. No elements need to be shifted as elements are added or removed. This approach requires, however, that we carefully manage the values of `front` and `rear`.

Let's look at another example. Figure 5.14 shows a circular array (drawn linearly) with a capacity of ten elements. Initially it is shown after elements A through H have been enqueued. It is then shown after the first four elements

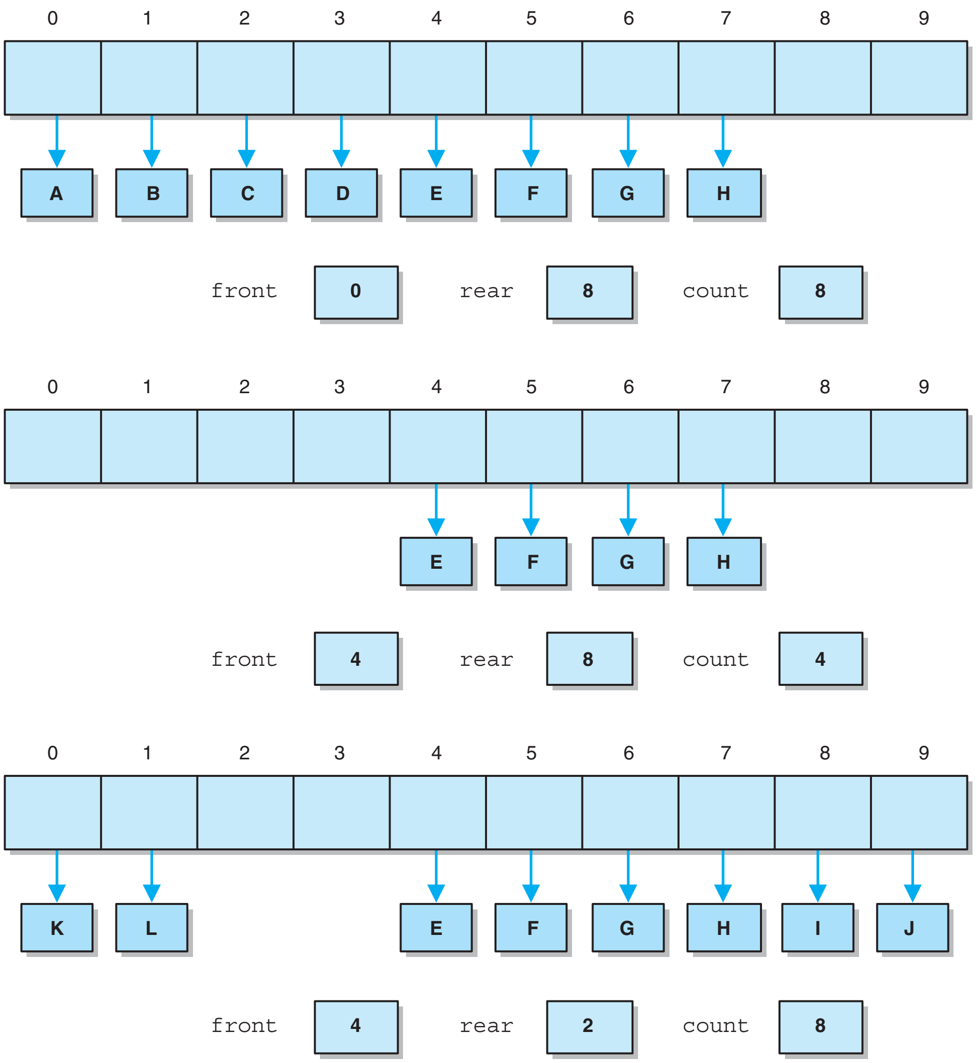


FIGURE 5.14 Changes in a circular array implementation of a queue

(A through D) have been dequeued. Finally, it is shown after elements I, J, K, and L have been enqueued, which causes the queue to wrap around the end of the array.

The header, class-level data, and constructors for our circular array implementation of a queue are provided for context:

```
/**
 * CircularArrayQueue represents an array implementation of a queue in
 * which the indexes for the front and rear of the queue circle back to 0
 * when they reach the end of the array.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0 08/12/08
 */

package jss2;

import jss2.exceptions.*;
import java.util.Iterator;

public class CircularArrayQueue<T> implements QueueADT<T>
{
  private final int DEFAULT_CAPACITY = 100;
  private int front, rear, count;
  private T[] queue;

  /**
   * Creates an empty queue using the default capacity.
   */

  public CircularArrayQueue()
  {
    front = rear = count = 0;
    queue = (T[]) (new Object[DEFAULT_CAPACITY]);
  }

  /**
   * Creates an empty queue using the specified capacity.
   *
   * @param initialCapacity  the integer representation of the initial
   *                         size of the circular array queue
   */

  public CircularArrayQueue (int initialCapacity)
  {
    front = rear = count = 0;
    queue = ( (T[])(new Object[initialCapacity]) );
  }
```

The enqueue Operation

In general, after an element is enqueued, the value of rear is incremented. But when an enqueue operation fills the last cell of the array (at the largest index), the value of rear must be set to 0, indicating that the next element should be stored at index 0. The appropriate update to the value of rear can be accomplished in one calculation by using the remainder operator (%). Recall that the remainder operator returns the remainder after dividing the first operand by the second. Therefore, if queue is the name of the array storing the queue, the following line of code will update the value of rear appropriately:

```
rear = (rear+1) % queue.length;
```

Let's try this calculation, assuming we have an array of size 10. If rear is currently 5, it will be set to 6%10, or 6. If rear is currently 9, it will be set to 10%10 or 0. Try this calculation using various situations to see that it works no matter how big the array is.

Given this strategy, the enqueue operation can be implemented as follows:

```
/**
 * Adds the specified element to the rear of this queue, expanding
 * the capacity of the queue array if necessary.
 *
 * @param element the element to add to the rear of the queue
 */

public void enqueue (T element)
{
   if  (size() == queue.length)
         expandCapacity();

   queue[rear] = element;
   rear = (rear+1) % queue.length;
   count++;
}
```

Note that this implementation strategy can still allow the array to reach capacity. As with any array-based implementation, all cells in the array may become filled. This implies that the rear of the queue has "caught up" to the front of the queue. To add another element, the array would have to be enlarged. Keep in mind, however, that the elements of the existing array must be copied into the new array in their proper order in the queue, not necessarily the order in which they appear in the current array. This makes the expandCapacity method slightly more complex than the one we used for stacks:

```
/**
 * Creates a new array to store the contents of this queue with
 * twice the capacity of the old one.
 */

public void expandCapacity()
{
  T[] larger = (T[])(new Object[queue.length *2]);

  for(int scan=0; scan < count; scan++)
  {
    larger[scan] = queue[front];
    front=(front+1) % queue.length;
  }
  front = 0;
  rear = count;
  queue = larger;
}
```

The dequeue Operation

Likewise, after an element is dequeued, the value of front is incremented. After enough dequeue operations, the value of front will reach the last index of the array. After removing the element at the largest index, the value of front must be set to 0 instead of being incremented. The same calculation we used to set the value of rear in the enqueue operation can be used to set the value of front in the dequeue operation:

```
/**
 * Removes the element at the front of this queue and returns a
 * reference to it. Throws an EmptyCollectionException if the
 * queue is empty.
 *
 * @return                               the reference to the element at the front
 *                                       of the queue that was removed
 * @throws EmptyCollectionException     if an empty collections exception occurs
 */
```

```
public T dequeue() throws EmptyCollectionException
{
  if  (isEmpty())
      throw new EmptyCollectionException ("queue");

  T result = queue[front];
  queue[front] = null;
  front = (front+1) % queue.length;
  count--;

  return result;
}
```

Other Operations

Operations such as `toString` become a bit more complicated using this approach because the elements are not stored starting at index 0 and may wrap around the end of the array. These methods have to take the current situation into account. All of the other operations for a circular array queue are left as programming projects.

Summary of Key Concepts

- Queue elements are processed in a FIFO manner—the first element in is the first element out.
- A queue is a convenient collection for storing a repeating code key.
- Simulations are often implemented using queues to represent waiting lines.
- A linked implementation of a queue is facilitated by references to the first and last elements of the linked list.
- The `enqueue` and `dequeue` operations work on opposite ends of the collection.
- Because queue operations modify both ends of the collection, fixing one end at index 0 requires that elements be shifted.
- The shifting of elements in a noncircular array implementation creates an O(n) complexity.
- Treating arrays as circular eliminates the need to shift elements in an array queue implementation.

Self-Review Questions

SR 5.1 What is the difference between a queue and a stack?

SR 5.2 What are the five basic operations on a queue?

SR 5.3 What are some of the other operations that might be implemented for a queue?

SR 5.4 Is it possible for the `front` and `rear` references in a linked implementation to be equal?

SR 5.5 Is it possible for the `front` and `rear` references in a circular array implementation to be equal?

SR 5.6 Which implementation has the worst time complexity?

SR 5.7 Which implementation has the worst space complexity?

Exercises

EX 5.1 Hand trace a queue X through the following operations:

```
X.enqueue(new Integer(4));
X.enqueue(new Integer(1));
```

```
Object Y = X.dequeue();
X.enqueue(new Integer(8));
X.enqueue(new Integer(2));
X.enqueue(new Integer(5));
X.enqueue(new Integer(3));
Object Y = X.dequeue();
X.enqueue(new Integer(4));
X.enqueue(new Integer(9));
```

EX 5.2 Given the resulting queue X from Exercise 5.1, what would be the result of each of the following?

a. `X.front();`

b. `Y = X.dequeue();`
   `X.front();`

c. `Y = X.dequeue();`

d. `X.front();`

EX 5.3 What would be the time complexity of the `size` operation for each of the implementations if there were not a `count` variable?

EX 5.4 Under what circumstances could the `front` and `rear` references be equal for each of the implementations?

EX 5.5 Hand trace the ticket counter problem for 22 customers and 4 cashiers. Graph the total process time for each person. What can you surmise from these results?

EX 5.6 Compare and contrast the `enqueue` method of the `LinkedQueue` class to the `push` method of the `LinkedStack` class from Chapter 4.

EX 5.7 Describe two different ways the `isEmpty` method of the `LinkedQueue` class could be implemented.

EX 5.8 Name five everyday examples of a queue other than those discussed in this chapter.

EX 5.9 Explain why the array implementation of a stack does not require elements to be shifted but the noncircular array implementation of a queue does.

EX 5.10 Suppose the `count` variable was not used in the `CircularArrayQueue` class. Explain how you could use the values of `front` and `rear` to compute the number of elements in the list.

Programming Projects

PP 5.1 Complete the implementation of the `LinkedQueue` class presented in this chapter. Specifically, complete the implementations of the `first`, `isEmpty`, `size`, and `toString` methods.

PP 5.2 Complete the implementation of the `CircularArrayQueue` class described in this chapter, including all methods.

PP 5.3 Write a version of the `CircularArrayQueue` class that grows the list in the opposite direction from which the version described in this chapter grows the list.

PP 5.4 All of the implementations in this chapter use a `count` variable to keep track of the number of elements in the queue. Rewrite the linked implementation without a `count` variable.

PP 5.5 All of the implementations in this chapter use a `count` variable to keep track of the number of elements in the queue. Rewrite the circular array implementation without a `count` variable.

PP 5.6 A data structure called a deque (pronounced like "deck") is closely related to a queue. The name *deque* stands for double-ended queue. The difference between the two is that with a deque, you can insert or remove from either end of the queue. Implement a deque using arrays.

PP 5.7 Implement the deque from Programming Project 5.6 using links. (*Hint:* Each node will need a `next` and a `previous` reference.)

PP 5.8 Create a graphical application that provides buttons for enqueue and dequeue from a queue, a text field to accept a string as input for enqueue, and a text area to display the contents of the queue after each operation.

PP 5.9 Create a system using a stack and a queue to test whether a given string is a palindrome (i.e., the characters read the same forward or backward).

PP 5.10 Create a system to simulate vehicles at an intersection. Assume that there is one lane going in each of four directions, with stoplights facing each direction. Vary the arrival average of vehicles in each direction and the frequency of the light changes to view the "behavior" of the intersection.

Answers to Self-Review Questions

SRA 5.1 A queue is a first in, first out (FIFO) collection, whereas a stack is a last in, first out (LIFO) collection.

SRA 5.2 The basic queue operations are:

`enqueue`—adds an element to the end of the queue

`dequeue`—removes an element from the front of the queue

`first`—returns a reference to the element at the front of the queue

`isEmpty`—returns true if the queue is empty, returns false otherwise

SRA 5.3 `makeEmpty()`, `destroy()`, `full()`

SRA 5.4 Yes, it happens when the queue is empty (both `front` and `rear` are null) and when there is only one element on the queue.

SRA 5.5 Yes, it can happen under two circumstances: when the queue is empty, and when the queue is full.

SRA 5.6 The noncircular array implementation with an O(n) `dequeue` or `enqueue` operation has the worst time complexity.

SRA 5.7 Both of the array implementations waste space for unfilled elements in the array. The linked implementation uses more space per element stored.

This page intentionally left blank

Lists

6

The concept of a list is inherently familiar to us. We make "to-do" lists, lists of items to buy at the grocery store, and lists of friends to invite to a party. We may number the items in a list or we may keep them in alphabetical order. For other lists we may keep the items in a particular order that simply makes the most sense to us. This chapter explores the concept of a list collection and some ways they can be managed.

CHAPTER OBJECTIVES

- Examine list processing and various ordering techniques
- Define a list abstract data type
- Introduce the concept of an iterator
- Examine polymorphism via interfaces
- Demonstrate how a list can be used to solve problems
- Examine various list implementations
- Compare list implementations

6.1 A List ADT

There are three types of list collections:

- *Ordered lists*, whose elements are ordered by some inherent characteristic of the elements
- *Unordered lists*, whose elements have no inherent order but are ordered by their placement in the list
- *Indexed lists*, whose elements can be referenced using a numeric index

KEY CONCEPT
List collections can be categorized as ordered, unordered, and indexed.

KEY CONCEPT
The elements of an ordered list have an inherent relationship defining their order.

KEY CONCEPT
The elements of an unordered list are kept in whatever order the client chooses.

An ordered list is based on some particular characteristic of the elements in the list. For example, you may keep a list of people ordered alphabetically by name, or you may keep an inventory list ordered by part number. The list is sorted based on some key value. Any element added to an ordered list has a proper location in the list, given its key value and the key values of the elements already in the list. Figure 6.1 shows a conceptual view of an ordered list, in which the elements are ordered by an integer key value. Adding a value to the list involves finding the new element's proper, sorted position among the existing elements.

The placement of elements in an unordered list is not based on any inherent characteristic of the elements. Don't let the name mislead you. The elements in an unordered list are kept in a particular order, but that order is not based on the elements themselves. The client using the list determines the order of the elements. Figure 6.2 shows a conceptual view of an unordered list. A new element can be put on the front or rear of the list, or it can be inserted after a particular element already in the list.

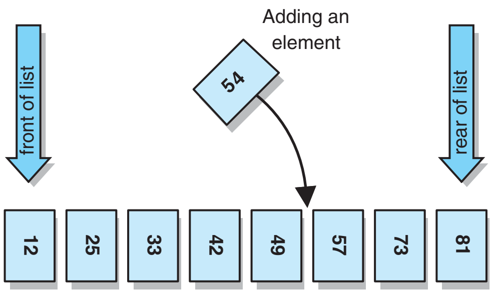


FIGURE 6.1 A conceptual view of an ordered list

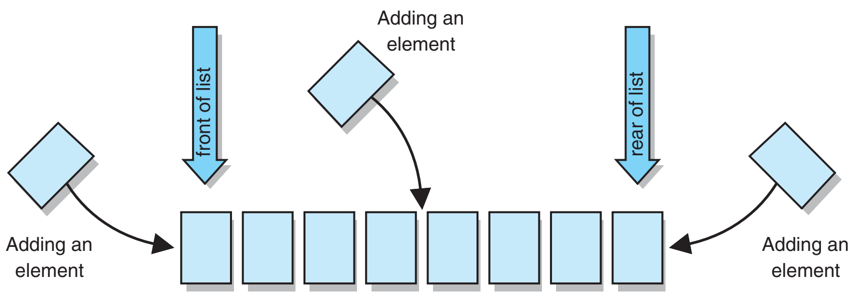


FIGURE 6.2 A conceptual view of an unordered list

An indexed list is similar to an unordered list in that there is no inherent relationship among the elements that determines their order in the list. The client using the list determines the order of the elements. However, in addition, each element can be referenced by a numeric index that begins at 0 at the front of the list and continues contiguously until the end of the list. Figure 6.3 shows a conceptual view of an indexed list. A new element can be inserted into the list at any position, including at the front or rear of the list. Every time a change occurs in the list, the indexes are adjusted to stay in order and contiguous.

Note the primary difference between an indexed list and an array: An indexed list keeps its indexes contiguous. If an element is removed, the positions of other elements "collapse" to eliminate the gap. When an element is inserted, the indexes of other elements are shifted to make room. The Java Collections API implements three different varieties of indexed lists, and we will explore these further

KEY CONCEPT
An indexed list maintains a contiguous numeric index range for its elements.

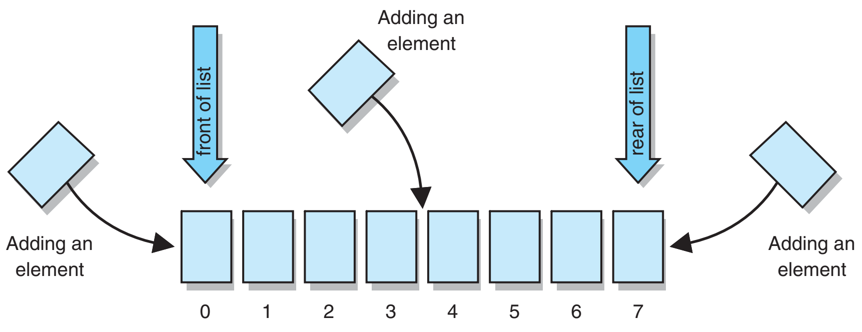


FIGURE 6.3 A conceptual view of an indexed list

Operation	Description
`removeFirst`	Removes the first element from the list.
`removeLast`	Removes the last element from the list.
`remove`	Removes a particular element from the list.
`first`	Examines the element at the front of the list.
`last`	Examines the element at the rear of the list.
`contains`	Determines if the list contains a particular element.
`isEmpty`	Determines if the list is empty.
`size`	Determines the number of elements on the list.

FIGURE 6.4 The common operations on a list

in Section 6.6. We will focus the majority of our discussion for now on ordered and unordered lists.

Keep in mind that these are conceptual views of lists. As with any collection, they can be implemented in many ways. The implementations of these lists don't even have to keep the elements in the order that their conceptual view indicates, though that may be easiest.

KEY CONCEPT
Many common operations can be defined for all list types. The differences between them stem from how elements are added.

There is a set of operations that is common to both ordered and unordered lists. These common operations are shown in Figure 6.4. They include operations to remove and examine elements, as well as classic operations such as `isEmpty` and `size`. The `contains` operation is also supported by both list types, which allows the user to determine if a list contains a particular element.

Note that for the first time, we are examining a collection that conceptually allows us to access elements in the middle of the list. With stacks, we were only able to operate on one end, the top, of the collection. With queues, we could add to one end and remove from the other. With lists, we are able to add, remove, or view any element in the list. For this reason, lists include an iterator.

Iterators

KEY CONCEPT
An iterator is an object that provides a means to iterate over a collection.

An *iterator* is an object that provides the means to iterate over a collection. That is, it provides methods that allow the user to acquire and use each element in a collection in turn. Most collections provide one or more ways to iterate over their elements. In the case of the `ListADT` interface, we define a method called `iterator` that returns an `Iterator` object.

The `Iterator` interface is defined in the Java standard class library. The two primary abstract methods defined in the `Iterator` interface are:

- `hasNext`, which returns true if there are more elements in the iteration
- `next`, which returns the next element in the iteration

The `iterator` method of the `ListADT` interface returns an object that implements this interface. The user can then interact with that object, using the `hasNext` and `next` methods, to access the elements in the list.

Note that there is no assumption about the order in which an `Iterator` object delivers the elements from the collection. In the case of a list, there is a linear order to the elements, so the iterator would likely follow that order. In other cases, an iterator may follow a different order that makes sense for that collection.

Another issue surrounding the use of iterators is what happens if the collection is modified while the iterator is in use. Most of the collections in the Java Collections API are implemented to be *fail-fast*. This simply means that they will, or should, throw an exception if the collection is modified while the iterator is in use. However, the documentation regarding these collections is very explicit that this behavior cannot be guaranteed. We will illustrate a variety of alternative possibilities for iterator construction throughout the examples in the book. These possibilities include creating iterators that allow concurrent modification and reflect those changes in the iteration, and creating iterators that iterate over a snapshot of the collection for which concurrent modifications have no impact. By including an `Iterator` method in our collection we also make the collection `Iterable` or in other words, we implement the `Iterable` interface. This means that, like we do with arrays, we can use the iterator version of a for loop: `for each . . .`

Adding Elements to a List

The differences between ordered and unordered lists generally center on how elements are added to the list. In an ordered list, we need only specify the new element to add. Its position in the list is based on its key value. This operation is shown in Figure 6.5.

Operation	Description
`add`	Adds an element to the list.

FIGURE 6.5 The operation particular to an ordered list

Operation	Description
`addToFront`	Adds an element to the front of the list.
`addToRear`	Adds an element to the rear of the list.
`addAfter`	Adds an element after a particular element already in the list.

FIGURE 6.6 The operations particular to an unordered list

An unordered list supports three variations of the `add` operation. Elements can be added to the front or rear of the list, or after a particular element that is already in the list. These operations are shown in Figure 6.6.

DESIGN FOCUS

Is it possible that a list could be both an ordered list and an indexed list? Possible perhaps but not very meaningful. If a list were both ordered and indexed, what would happen if a client application attempted to add an element at a particular index or change an element at a particular index such that it is not in the proper order? Which rule would have precedence, index position or order?

Conceptually, the operations particular to an indexed list make use of its ability to reference elements by their index. A new element can be inserted into the list at a particular index, or it can be added to the rear of the list without specifying an index at all. Note that if an element is inserted or removed, the elements at higher indexes are either shifted up to make room or shifted down to close the gap. Alternatively, the element at a particular index can be set, which overwrites the element currently at that index and therefore does not cause other elements to shift. In addition, as we will explore later in this chapter, indexed lists provide operations to retrieve an element at a particular index and to determine the index of an element in the list.

We can capitalize on the fact that both ordered lists and unordered lists share a common set of operations. These operations need to be defined only once. Therefore, we will define three list interfaces: one with the common operations and two with the operations particular to each list type. Inheritance can be used with interfaces just as it can with classes. The interfaces of the particular list types extend the common list definition. This relationship among the interfaces is shown in Figure 6.7.

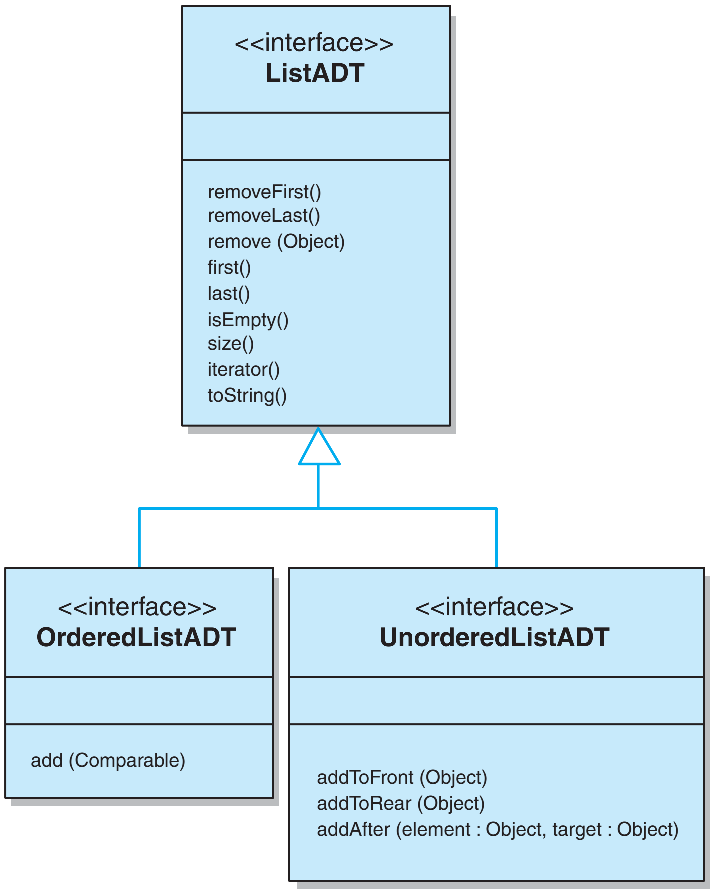


FIGURE 6.7 The various list interfaces

When interfaces are inherited, the child interface contains all abstract methods defined in the parent. Therefore, any class implementing a child interface must implement all methods from both the parent and the child.

KEY CONCEPT
Interfaces can be used to derive other interfaces. The child interface contains all abstract methods of the parent.

Interfaces and Polymorphism

Up to this point, we have been able to simply store a generic type <T> in our stack and queue collections. The generic type <T> is then replaced with a specific type at the time a collection is instantiated. We have not placed any restrictions on what types could be stored in our collections. However, in dealing with ordered lists, we must place a restriction on the type so that only classes that implement the `Comparable` interface can be stored in an ordered list. This is necessary so that we can use the classes own `compareTo` method to order the elements in the list. This is sometimes referred to as the *natural ordering* of the elements. In Chapter 3, we discussed the concept of polymorphism via inheritance. Let's examine how interfaces can be used to create polymorphism as well.

As we have seen, a class name is used to declare the type of an object reference variable. Similarly, an interface name can be used as the type of a reference variable as well. An interface reference variable can be used to refer to any object of any class that implements that interface.

Suppose we declare an interface called `Speaker` as follows:

```
public interface Speaker
{
   public void speak();
   public void announce (String str);
}
```

KEY CONCEPT
An interface name can be used to declare an object reference variable. An interface reference can refer to any object of any class that implements the interface.

The interface name, `Speaker`, can now be used to declare an object reference variable:

```
Speaker current;
```

The reference variable `current` can be used to refer to any object of any class that implements the `Speaker` interface. For example, if we define a class called `Philosopher` such that it implements the `Speaker` interface, we could then assign a `Philosopher` object to a `Speaker` reference:

```
current = new Philosopher();
```

This assignment is valid because a `Philosopher` is, in fact, a `Speaker`.

The flexibility of an interface reference allows us to create polymorphic references. As we saw in Chapter 3, by using inheritance, we can create a polymorphic reference that can refer to any one of a set of objects related by inheritance. Using interfaces, we can create similar polymorphic references, except that the objects being referenced are related by implementing the same interface instead of being related by inheritance.

KEY CONCEPT
Interfaces allow us to make polymorphic references in which the method that is invoked is based on the particular object being referenced at the time.

For example, if we create a class called `Dog` that also implements the `Speaker` interface, it too could be assigned to a `Speaker` reference variable. The same reference, in fact, could at one point refer to a `Philosopher` object, and then later refer to a `Dog` object. The following lines of code illustrate this:

```
Speaker guest;
guest = new Philosopher();
guest.speak();
guest = new Dog();
guest.speak();
```

In this code, the first time the speak method is called, it invokes the speak method defined in the Philosopher class. The second time it is called, it invokes the speak method of the Dog class. As with polymorphic references via inheritance, it is not the type of the reference that determines which method gets invoked, but rather the type of the object that the reference points to at the moment of invocation.

Note that when we are using an interface reference variable, we can invoke only the methods defined in the interface, even if the object it refers to has other methods to which it can respond. For example, suppose the Philosopher class also defined a public method called pontificate. The second line of the following code would generate a compiler error, even though the object can in fact respond to the pontificate method:

```
Speaker special = new Philosopher();
special.pontificate(); // generates a compiler error
```

The problem is that the compiler can determine only that the object is a Speaker, and therefore can guarantee only that the object can respond to the speak and announce methods. Because the reference variable special could refer to a Dog object (which cannot pontificate), it does not allow the reference. If we know that in a particular situation such an invocation is valid, we can cast the object into the appropriate reference so that the compiler will accept it:

```
((Philosopher) special).pontificate();
```

Similar to polymorphic references based on inheritance, an interface name can be used as the type of a method parameter. In such situations, any object of any class that implements the interface can be passed into the method. For example, the following method takes a Speaker object as a parameter. Therefore, both a Dog object and a Philosopher object can be passed into it in separate invocations.

```
public void sayIt (Speaker current)
{
   current.speak();
}
```

Given this discussion, could we simply make the reference type of all of the elements in our ordered list Comparable? This would mean that any object of any class that implements the Comparable interface could be stored in any instance of our ordered list collection. However, simply having objects that have implemented the `Comparable` interface does not mean that those objects can necessarily be compared to each other, only that they can be compared to other objects of the same type. This is certainly not what we intended.

The `Comparable` interface is generic so what if we tried the following as a generic type for our collection:

```
<T extends Comparable<T>>
```

This still causes a problem. Assume for a moment that both `Philosopher` and `Dog` are subclasses of a class `Mammal` and that the `Mammal` class implements the `Comparable<Mammal>` interface. This results in both philosophers and dogs being comparable to other mammals but not exclusively to members of their own class. In other words, using `<T extends Comparable<T>>` would not allow us to create an ordered list of philosophers because philosophers are comparable to mammals, not just to other philosophers.

A more comprehensive solution, though not necessarily a more intellectually satisfying solution is to write the generic type as:

```
<T extends Comparable<? super T>>
```

This will include both the case where `T` implements `Comparable<T>` and the case where some superclass of `T` implements `Comparable`.

The implementers of the Java language chose a different alternative for the types for the ordered collections in the Java Collections API. In these cases, the type is simply implemented as a generic type `<T>`, and the methods that add elements to the collection will throw a `ClassCastException` if the element being added cannot be compared to the elements in the collection. We will adopt that same approach for our ordered list collection.

Listings 6.1 through 6.3 show the Java interfaces corresponding to the UML diagram in Figure 6.7.

Before exploring how these various kinds of lists can be implemented, let's first see how they might be used.

6.2 Using Ordered Lists: Tournament Maker

Sporting tournaments, such as the NCAA basketball tournament or a championship tournament at a local bowling alley, are often organized or seeded by the number of wins achieved during the regular season. Ordered lists can be used to help organize the tournament play. An ordered list can be used to store teams ordered by number of wins. To form the match-ups for the first round of the

LISTING 6.1

```java
/**
 * ListADT defines the interface to a general list collection. Specific
 * types of lists will extend this interface to complete the
 * set of necessary operations.
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 08/13/08
 */

package jss2;
import java.util.Iterator;

public interface ListADT<T> extends Iterable<T>
{
   /**
    * Removes and returns the first element from this list.
    *
    * @return the first element from this list
    */
   public T removeFirst ();

   /**
    * Removes and returns the last element from this list.
    *
    * @return the last element from this list
    */
   public T removeLast ();

   /**
    * Removes and returns the specified element from this list.
    *
    * @param element the element to be removed from the list
    */
   public T remove (T element);

   /**
    * Returns a reference to the first element in this list.
    *
    * @return a reference to the first element in this list
    */
   public T first ();
```

LISTING 6.1 *continued*

```
   /**
    * Returns a reference to the last element in this list.
    *
    * @return a reference to the last element in this list
    *//
   public T last ();

   /**
    * Returns true if this list contains the specified target element.
    *
    * @param target  the target that is being sought in the list
    * @return        true if the list contains this element
    */
   public boolean contains (T target);

   /**
    * Returns true if this list contains no elements.
    *
    * @return true if this list contains no elements
    */
   public boolean isEmpty();

   /**
    * Returns the number of elements in this list.
    *
    * @return the integer representation of number of elements in this list
    */
   public int size();

   /**
    * Returns an iterator for the elements in this list.
    *
    * @return an iterator over the elements in this list
    */
   public Iterator<T> iterator();

   /**
    * Returns a string representation of this list.
    *
    * @return a string representation of this list
    */
   public String toString();
}
```

LISTING 6.2

```
/**
 * OrderedListADT defines the interface to an ordered list collection. Only
 * comparable elements are stored, kept in the order determined by
 * the inherent relationship among the elements.
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 08/13/08
 */

package jss2;

public interface OrderedListADT<T> extends ListADT<T>
{
   /**
    * Adds the specified element to this list at the proper location
    *
    * @param element the element to be added to this list
    */
   public void add (T element);
}
```

LISTING 6.3

```
/**
 * UnorderedListADT defines the interface to an unordered list collection.
 * Elements are stored in any order the user desires.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 08/13/08
 */

package jss2;

public interface UnorderedListADT<T> extends ListADT<T>
{

   /**
    * Adds the specified element to the front of this list.
    *
```

LISTING 6.3 *continued*

```
     * @param element the element to be added to the front of this list
     */
    public void addToFront (T element);

    /**
     * Adds the specified element to the rear of this list.
     *
     * @param element the element to be added to the rear of this list
     */
    public void addToRear (T element);

    /**
     * Adds the specified element after the specified target.
     *
     * @param element the element to be added after the target
     * @param target the target is the item that the element will be added
     *        after
     */
    public void addAfter (T element, T target);
}
```

KEY CONCEPT
An ordered list is a convenient collection to use when creating a tournament schedule.

tournament, teams can be selected from the front and back of the list in pairs.

For example, consider the eight bowling teams listed in Figure 6.8. This table indicates the number of wins each team achieved during the regular season.

Team Name	Wins
Scorecards	10
Gutterballs	9
KingPins	8
PinDoctors	7
Spares	5
Splits	4
Tenpins	3
Woodsplitters	2

FIGURE 6.8 Bowling league team names and number of wins

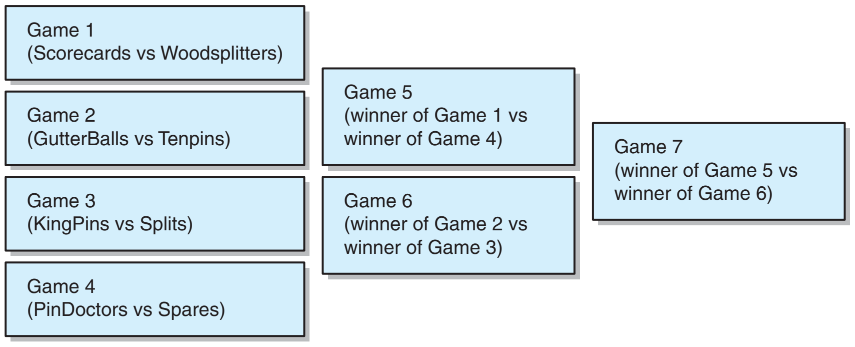

Game 1
(Scorecards vs Woodsplitters)
Game 2
(GutterBalls vs Tenpins)
Game 3
(KingPins vs Splits)
Game 4
(PinDoctors vs Spares)
Game 5
(winner of Game 1 vs winner of Game 4)
Game 6
(winner of Game 2 vs winner of Game 3)
Game 7
(winner of Game 5 vs winner of Game 6)

FIGURE 6.9 Sample tournament layout for a bowling league tournament

To create the first-round tournament matches, the teams would be stored in a list ordered by the number of wins. The first team on the list (the team with the best record) is removed from the list and matched up with the last team on the list (the team with the worst record) to form the first game of the tournament. The process is repeated, matching up the team with the next best record with the team with the next worst record to form the second game. This process continues until the list is empty. Interestingly, the same process would be used to form the second-round match-ups, only for the second round, the teams would be ordered by game number from the first round. For the third round, the teams would be ordered by game number from the second round. This process would continue with half as many games per round until only one game was left. Thus, from our example in Figure 6.8, we would end up with the tournament as laid out in Figure 6.9.

Creating a program to select the first-round tournament match-ups requires that we first create a class to represent the information we wish to store about the teams. This `Team` class needs to store both the name of the team and the number of wins. The `Team` class also needs to provide us with some sort of comparison operation. For this purpose, the `Team` class will implement the `Comparable` interface, thus providing a `compareTo` method. This method will return −1 if the first team has fewer wins than the second team, 0 if the two teams have the same number of wins, and 1 if the first team has more wins than the second team. As we will discuss in more detail in the next section, any class that we intend to store in our ordered list collection must implement the `Comparable` interface. Figure 6.10 illustrates the UML relationships among the classes used to solve this problem. Listing 6.4 shows the `Tournament` class, Listing 6.5 shows the `Team` class, and Listing 6.6 illustrates the `TournamentMaker` class.

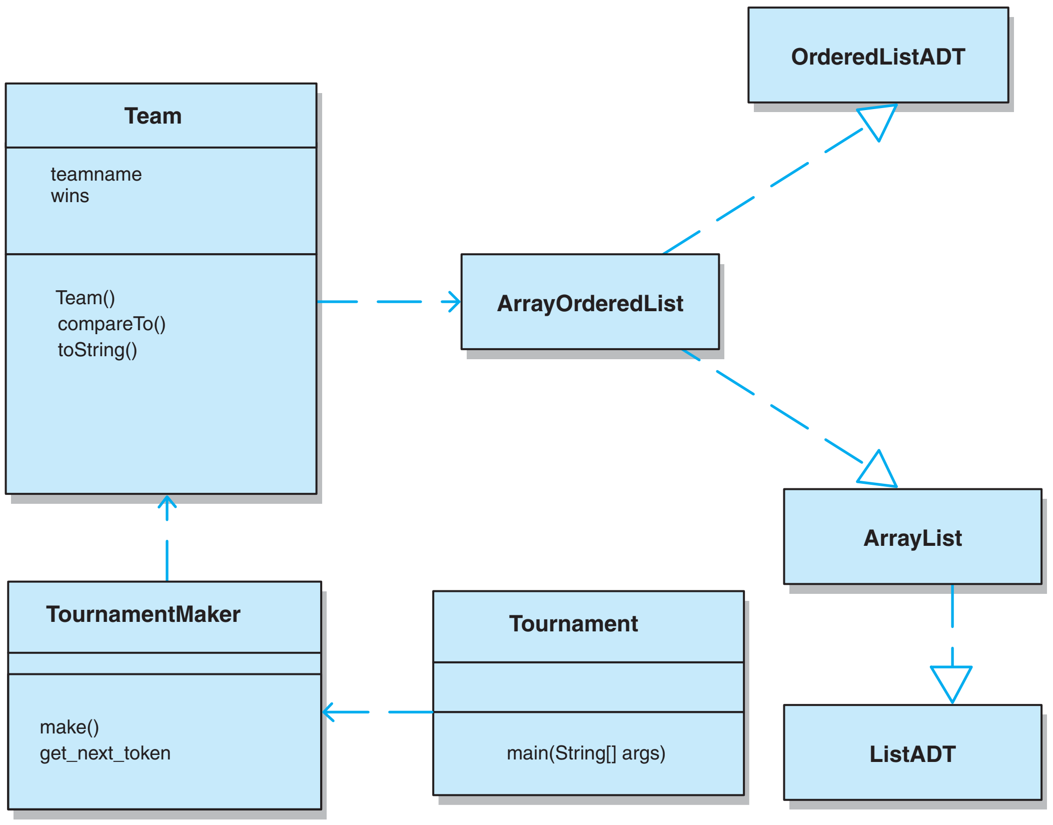


FIGURE 6.10 UML description of the `Tournament` class

LISTING 6.4

```
/**
 * Tournament is a driver for a program that demonstrates first round of
 * tournament team match-ups using an ordered list.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 08/12/08
 */

import java.io.*;

public class Tournament
{
```

LISTING 6.4 *continued*

```
   /**
    * Determines and prints the tournament organization.
    */
   public static void main (String[] args ) throws IOException
   {
     TournamentMaker temp = new TournamentMaker();
     temp.make();
   }
}
```

LISTING 6.5

```
/**
 * Team represents a team.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 08/12/08
 */

import java.util.*;

class Team implements Comparable<Team>
{
   public String teamName;
   private int wins;

   /**
    * Sets up this team with the specified information.
    *
    * @param name     the string representation of the name of this team
    * @param numWins  the integer representation of the number of wins for this
    *                 team
    */
   public Team (String name, int numWins)
   {
      teamName = name;
      wins = numWins;
   }
```

LISTING 6.5 *continued*

```
   /**
    * Returns the name of the given team.
    *
    * @return the string representation of the name of this team
    */
   public String getName ()
   {
      return teamName;
   }

   /**
    * Compares number of wins of this team to another team. Returns
    * -1, 0, or 1 for less than, equal to, or greater than.
    *
    * @param other  the team to compare wins against this team
    * @return       the integer representation of the result of this comparison,
    * valid         values are -1. 0, 1, for less than, equal to, and
    *               greater than.
    */
   public int compareTo (Team other)
   {
      if (this.wins < other.wins)
         return -1;
      else
        if (this.wins == other.wins)
           return 0;
      else
         return 1;
   }

   /**
    * Returns the name of the team.
    *
    * @return the string representation of the name of this team
    */
   public String toString()
   {
     return teamName;
   }
}
```

LISTING 6.6

```
/**
 * TournamentMaker demonstrates first round of tournament team match-ups
 * using an ordered list.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 08/12/08
 */

import jss2.*;
import jss2.exceptions.*;
import java.util.Scanner;
import java.io.*;
import java.lang.*;

public class TournamentMaker
{

   /**
    * Determines and prints the tournament organization.
    *
    * @throws IOException if an IO exception is encountered
    */
   public void make ( ) throws IOException
   {
      ArrayOrderedList<Team> tournament = new ArrayOrderedList<Team>();
      String team1, team2, teamName;
      int numWins, numTeams = 0;
      double checkInput=-1;

      Scanner in = new Scanner(System.in);

      System.out.println("Tournament Maker\n");

      while (((numTeams % 2) != 0) || (numTeams == 2) || (checkInput!=0))
      {
         System.out.println ("Enter the number of teams (must be an even \n" +
                  "number valid for a single elimination tournament):");
         numTeams = in.nextInt();
         in.nextLine(); // advance beyond new line char

         /** checks if numTeams is valid for single elimination tournament */
         checkInput = (Math.log(numTeams)/Math.log(2)) % 1;
      }
```

LISTING 6.6 *continued*

```
      System.out.println ("\nEnter " + numTeams + " team names and number of wins.");
      System.out.println("Teams may be entered in any order.");
      for (int count=1; count <= numTeams; count++)
      {
         System.out.println("Enter team name: ");
         teamName = in.nextLine();
         System.out.println("Enter number of wins: ");
         numWins = in.nextInt();
         in.nextLine(); // advance beyond new line char
         tournament.add(new Team(teamName, numWins));
      }
      System.out.println("\nThe first round matchups are: ");

      for (int count=1; count <=(numTeams/2); count++)
      {
          team1 = (tournament.removeFirst()).getName();
          team2 = (tournament.removeLast()).getName();
          System.out.println ("Game " + count + " is " + team1 +
             " against " + team2);
          System.out.println ("with the winner to play the winner of game "
             + (((numTeams/2)+1) - count) + "\n");
      }
   }
}
```

6.3 Using Indexed Lists: The Josephus Problem

Flavius Josephus was a Jewish historian of the first century. Legend has it that he was one of a group of 41 Jewish rebels who decided to kill themselves rather than surrender to the Romans, who had them trapped. They decided to form a circle and to kill every third person until no one was left. Josephus, not wanting to die, calculated where he needed to stand so that he would be the last one alive and thus would not have to die. Thus was born a class of problems referred to as the Josephus problem. These problems involve finding the order of events when events in a list are not taken in order, but rather they are taken every ith element in a cycle until none remains.

KEY CONCEPT
The Josephus problem is a classic computing problem that is appropriately solved with indexed lists.

For example, suppose that we have a list of seven elements numbered from 1 to 7:

```
1 2 3 4 5 6 7
```

If we were to remove every third element from the list, the first element to be removed would be number 3, leaving the list:

```
1 2 4 5 6 7
```

The next element to be removed would be number 6, leaving the list:

```
1 2 4 5 7
```

The elements are thought of as being in a continuous cycle, so that when we reach the end of the list, we continue counting at the beginning. Therefore, the next element to be removed would be number 2, leaving the list:

```
1 4 5 7
```

The next element to be removed would be number 7, leaving the list:

```
1 4 5
```

The next element to be removed would be number 5, leaving the list:

```
1 4
```

The next to last element to be removed would be number 1, leaving the number 4 as the last element on the list.

Listing 6.7 illustrates a generic implementation of the Josephus problem, allowing the user to input the number of items in the list and the gap between elements. Note that the original list is placed in an indexed list. Each element is then

LISTING 6.7

```java
/**
* Josephus
*
* @author Dr. Lewis
* @author Dr. Chase
* @version 1.0, 08/13/08
*/

import java.util.ArrayList;
import java.util.Scanner;

public class Josephus
{

   /**
    * Continue around the circle eliminating every nth soldier
    * until all of the soldiers have been eliminated.
    */
   public static void main (String[] args)
   {
      int numPeople, gap, newGap, counter;
      ArrayList<Integer> list = new ArrayList<Integer>();
      Scanner in = new Scanner(System.in);
      // get the initial number of soldiers
      System.out.println("Enter the number of soldiers: ");
      numPeople = in.nextInt();
```

LISTING 6.7 *continued*

```
      in.nextLine();
      // get the gap between soldiers
      System.out.println("Enter the gap between soldiers: ");
      gap = in.nextInt();

      // load the initial list of soldiers
      for (int count=1; count <= numPeople; count++)
      {
         list.add(new Integer(count));
      }
      counter = gap - 1;
      newGap = gap;
      System.out.println("The order is: ");

      // Treating the list as circular, remove every nth element
      // until the list is empty
      while (!(list.isEmpty()))
      {
         System.out.println(list.remove(counter));
         numPeople = numPeople - 1;
         if (numPeople > 0)
            counter = (counter + gap - 1) % numPeople;
      }
   }
}
```

removed from the list one at a time by computing the next index position in the list to be removed. The one complication in this process is the computation of the next index position to be removed. This is particularly interesting since the list collapses on itself as elements are removed. For example, the element number 6 from our previous example should be the second element removed from the list. However, once element 3 has been removed from the list, element 6 is no longer in its original position. Instead of being at index position 5 in the list, it is now at index position 4. Figure 6.11 illustrates the UML for the `Josephus` program. Notice that we have chosen to use the `ArrayList` implementation from the Java Collections API, which is actually an indexed list implementation.

6.4 Implementing Lists: With Arrays

An array-based implementation of a list could fix one end of the list at index 0 and shift elements as needed. This is similar to our array-based implementation of

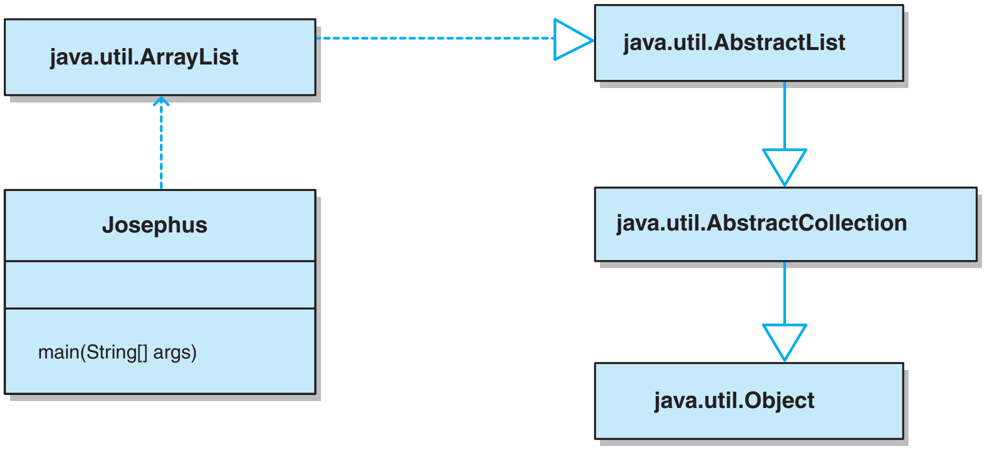


FIGURE 6.11 UML description of the Josephus program

a stack from Chapter 3 and to the first array-based implementation of a queue we discussed in Chapter 5. We dismissed that implementation as too inefficient for our queue implementation because of the need to shift elements in the array either on enqueue or dequeue. However, with lists, we will now insert elements into and remove elements from the middle of the list and thus shifting of elements cannot be avoided. We could still use a circular array approach as we did with our array-based queue implementation, but that will not eliminate the need to shift elements when adding or removing elements. That approach is left as a programming project.

Figure 6.12 shows an array implementation of a list with the front of the list fixed at index 0. The integer variable `rear` represents the number of elements in the list and the next available slot for adding an element to the rear of the list.

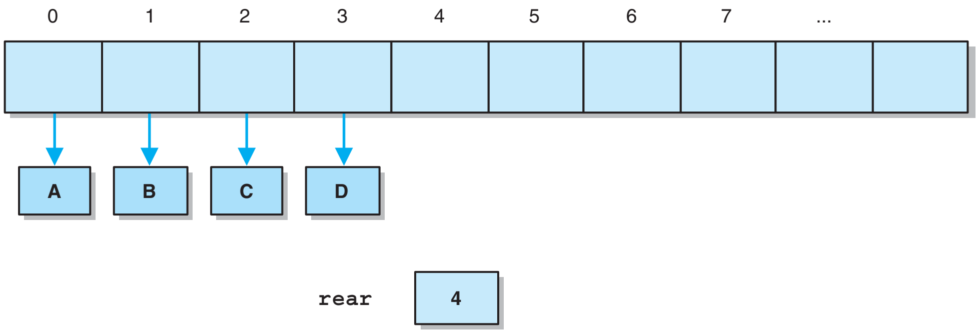


FIGURE 6.12 An array implementation of a list

Note that Figure 6.12 applies to both ordered and unordered lists. First we will explore the common operations.

The header and class-level data of the `ArrayList` class are listed here to provide context:

```
/**
 * ArrayList represents an array implementation of a list. The front of
 * the list is kept at array index 0. This class will be extended
 * to create a specific kind of list.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 08/13/08
 */

package jss2;
import jss2.exceptions.*;
import java.util.Iterator;

public class ArrayList<T> implements ListADT<T>, Iterable<T>
{
   protected final int DEFAULT_CAPACITY = 100;
   private final int NOT_FOUND = -1;
   protected int rear;
   protected T[] list;

   /**
    * Creates an empty list using the default capacity.
    */
   public ArrayList()
   {
      rear = 0;
      list = (T[])(new Object[DEFAULT_CAPACITY]);
   }

   /**
    * Creates an empty list using the specified capacity.
    *
    * @param initialCapacity the integer value of the size of the array list
    */
   public ArrayList (int initialCapacity)
   {
      rear = 0;
      list = (T[])(new Object[initialCapacity]);
   }
```

The remove Operation

This variation of the remove operation requires that we search for the element passed in as a parameter and remove it from the list if it is found. Then, elements at higher indexes in the array are shifted down in the list to fill in the gap. Consider what happens if the element to be removed is the first element in the list. In this case, there is a single comparison to find the element followed by n–1 shifts to shift the elements down to fill the gap. On the opposite extreme, what happens if the element to be removed is the last element in the list? In this case, we would require n comparisons to find the element and none of the remaining elements would need to be shifted. As it turns out, this implementation of the remove operation will always require exactly n comparisons and shifts and thus the operation is O(n). Note that if we were to use a circular array implementation, it would only improve the performance of the special case when the element to be removed is the first element. This operation can be implemented as follows:

```
/**
 * Removes and returns the specified element.
 *
 * @param element                   the element to be removed and returned
 *                                  from the list
 * @return                          the removed element
 * @throws ElementNotFoundException  if an element not found exception occurs
 */
public T remove (T element)
{
   T result;
   int index = find (element);

   if (index == NOT_FOUND)
      throw new ElementNotFoundException ("list");

   result = list[index];
   rear--;

   // shift the appropriate elements
   for (int scan=index; scan < rear; scan++)
      list[scan] = list[scan+1];

   list[rear] = null;

   return result;
}
```

The remove method makes use of a method called find, which finds the element in question, if it exists in the list, and returns its index. The find method returns a constant called NOT_FOUND if the element is not in the list. The NOT_FOUND constant is equal to −1 and is defined in the ArrayList class. If the element is not found, a NoSuchElementException is generated. If it is found, the elements at higher indexes are shifted down, the rear value is updated, and the element is returned.

The find method supports the implementation of a public operation on the list, rather than defining a new operation. Therefore, the find method is declared with private visibility. The find method can be implemented as follows:

```
/**
 * Returns the array index of the specified element, or the
 * constant NOT_FOUND if it is not found.
 *
 * @param target  the element that the list will be searched for
 * @return        the integer index into the array containing the target
 *                element, or the NOT_FOUND constant
 */
private int find (T target)
{
   int scan = 0, result = NOT_FOUND;
   boolean found = false;

   if (! isEmpty())
      while (! found && scan < rear)
         if (target.equals(list[scan]))
            found = true;
         else
            scan++;

   if (found)
      result = scan;

   return result;
}
```

Note that the find method relies on the equals method to determine if the target has been found. It's possible that the object passed into the method is an exact copy of the element being sought. In fact, it may be an alias of the element in the list. However, if the parameter is a separate object, it may not contain all aspects of the element being sought. Only the key characteristics on which the equals method is based are important.

The logic of the find method could have been incorporated into the remove method, though it would have made the remove method somewhat complicated. When appropriate, such support methods should be defined to keep each method readable. Furthermore, in this case, the find support method is useful in implementing the contains operation, as we will now explore.

DESIGN FOCUS

The overriding of the equals method and the implementation of the Comparable interface are excellent examples of the power of object-oriented design. We can create implementations of collections that can handle classes of objects that have not yet been designed as long as those objects provide a definition of equality and/or a method of comparison between objects of the class.

DESIGN FOCUS

Separating out private methods such as the find method in the ArrayList class provides multiple benefits. First, it simplifies the definition of the already complex remove method. Second, it allows us to use the find method to implement the contains operation as well as the addAfter method for an ArrayUnorderedList. Notice that the find method does not throw an ElementNotFound exception. It simply returns a value (–1), signifying that the element was not found. In this way, the calling routine can decide how to handle the fact that the element was not found. In the remove method, that means throwing an exception. In the contains method, that means returning false.

The contains Operation

The purpose of the contains operation is to determine if a particular element is currently contained in the list. As we discussed, we can use the find support method to create a fairly straightforward implementation:

```
/**
 * Returns true if this list contains the specified element.
 *
 * @param target  the element that the list is searched for
 * @return        true if the target is in the list, false if otherwise
 */
public boolean contains (T target)
{
   return (find(target) != NOT_FOUND);
}
```

If the target element is not found, the `contains` method returns false. If it is found, it returns true. A carefully constructed `return` statement ensures the proper return value. Because this method is performing a linear search of our list, our worst case will be that the element we are searching for is not in the list. This case would require n comparisons. We would expect this method to require, on average, n/2 comparisons, which results in the operation being O(n).

The `iterator` Operation

We have emphasized the idea thus far that we should reuse code whenever possible and design our solutions such that we can reuse them. The `iterator` operation is an excellent example of this philosophy. It would be possible to create an `iterator` method specifically for the array implementation of a list. However, instead we have created a general `ArrayIterator` class that will work with any array-based implementation of any collection. The `iterator` method for the array implementation of a list creates an instance of the `ArrayIterator` class. Listing 6.8 shows the `ArrayIterator` class.

```
/**
 * Returns an iterator for the elements currently in this list.
 *
 * @return an iterator for the elements in this list
 */
public Iterator<T> iterator()
{
   return new ArrayIterator<T> (list, rear);
}
```

The remaining common list operations are left as programming projects. Let's turn our attention now to the operations that are particular to a specific type of list.

The `add` Operation for an Ordered List

The `add` operation is the only way an element can be added to an ordered list. No location is specified in the call because the elements themselves determine their order. Very much like the `remove` operation, the `add` operation will require a combination of comparisons and shifts: comparisons to find the correct location in the list

LISTING 6.8

```
/**
 * ArrayIterator represents an iterator over the elements of an array.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 08/12/08
 */

package jss2;
import java.util.*;

public class ArrayIterator<T> implements Iterator<T>
{
   private int count;        // the number of elements in the collection
   private int current;      // the current position in the iteration
   private T[] items;

   /**
    * Sets up this iterator using the specified items.
    *
    * @param collection  the collection to create the iterator for
    * @param size        the size of the collection
    */
   public ArrayIterator (T[] collection, int size)
   {
      items = collection;
      count = size;
      current = 0;
   }

   /**
    * Returns true if this iterator has at least one more element
    * to deliver in the iteration.
    *
    * @return  true if this iterator has at least one more element to deliver
    *          in the iteration
    */
   public boolean hasNext()
   {
      return (current < count);
   }
```

LISTING 6.8 *continued*

```
    /**
     * Returns the next element in the iteration. If there are no
     * more elements in this iteration, a NoSuchElementException is
     * thrown.
     *
     * @return the next element in the iteration
     * @throws NoSuchElementException if an element not found exception occurs
     */
    public T next()
    {
       if (! hasNext())
          throw new NoSuchElementException();

    current++;

    return items[current - 1];
    }

    /**
     * The remove operation is not supported in this collection.
     *
     * @throws UnsupportedOperationException  if an unsupported operation
     *                                        exception occurs
     */
    public void remove() throws UnsupportedOperationException
    {
       throw new UnsupportedOperationException();
    }
}
```

and then shifts to open a position for the new element. Looking at the two extremes, if the element to be added to the list belongs at the front of the list, that will require one comparison and then the other n - 1 elements in the list will need to be shifted. If the element to be added belongs at the rear of the list, then this will require n comparisons and none of the other elements in the list will need to be shifted. Like the `remove` operation, the `add` operation will require n comparisons and shifts each time it is executed and thus the operation is O(n). The `add` operation can be implemented as follows:

```
/**
 * Adds the specified Comparable element to this list, keeping
 * the elements in sorted order.
 *
 * @param element the element to be added to this list
 */

public void add (T element)
{
   if (size() == list.length)
      expandCapacity();

   Comparable<T> temp = (Comparable<T>)element;

   int scan = 0;
     while (scan < rear && temp.compareTo(list[scan]) > 0)
        scan++;

   for (int scan2=rear; scan2 > scan; scan2--)
      list[scan2] = list[scan2-1];

   list[scan] = element;
   rear++;
}
```

Note that only `Comparable` objects can be stored in an ordered list. If an attempt is made to add a non-`Comparable` object to an `ArrayOrderedList`, a `ClassCastException` will result.

KEY CONCEPT
Only `Comparable` objects can be stored in an ordered list.

Recall that the `Comparable` interface defines the `compareTo` method that returns a negative, zero, or positive integer value if the executing object is less than, equal to, or greater than the parameter, respectively.

The unordered and indexed versions of a list do not require that the elements they store be `Comparable`. It is a testament to object-oriented programming that the various classes that implement these list variations can exist in harmony despite these differences.

Operations Particular to Unordered Lists

The `addToFront` and `addToRear` operations are similar to operations from other collections and therefore left as programming projects. Keep in mind that the addToFront operation must shift the current elements in the list first to make room at index 0 for the new element. Thus we know that the addToFront operation will be O(n) because it requires n–1 elements to be shifted. Like the push operation on a stack, the addToRear operation will be O(1).

The addAfter Operation for an Unordered List

The addAfter operation accepts two parameters: one that represents the element to be added and one that represents the target element that determines the placement of the new element. The addAfter method must first find the target element, shift the elements at higher indexes to make room, and then insert the new element after it. Very much like the remove operation and the add operation for ordered lists, the addAfter method will require a combination of n comparisons and shifts and will be O(n).

```
/**
 * Adds the specified element after the specified target element.
 * Throws an ElementNotFoundException if the target is not found.
 *
 * @param element  the element to be added after the target element
 * @param target   the target that the element is to be added after
 */
public void addAfter (T element, T target)
{
   if (size() == list.length)
     expandCapacity();

   int scan = 0;
   while (scan < rear && !target.equals(list[scan]))
       scan++;
   if (scan == rear)
       throw new ElementNotFoundException ("list");

   scan++;
   for (int scan2=rear; scan2 > scan; scan2--)
      list[scan2] = list[scan2–1];

   list[scan] = element;
   rear++;
}
```

6.5 Implementing Lists: With Links

As we have seen with other collections, the use of a linked list is often another convenient way to implement a linear collection. The common operations that apply for ordered and unordered lists, as well as the particular operations for the both types, can be implemented with techniques similar to the ones that we have used before. We will examine a couple of the more interesting operations but will leave most of these as programming projects.

The class header, class-level data, and constructor for our `LinkedList` class are provided for context.

```java
/**
 * LinkedList represents a linked implementation of a list.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 08/13/08
 */

package jss2;
import jss2.exceptions.*;
import java.util.*;

public class LinkedList<T> implements ListADT<T>, Iterable<T>
{
   protected int count;
   protected LinearNode<T> head, tail;

   /**
    * Creates an empty list.
    */
   public LinkedList()
   {
      count = 0;
      head = tail = null;
   }
```

The `remove` Operation

The `remove` operation is part of the `LinkedList` class shared by both implementations: unordered and ordered lists. The `remove` operation consists of making sure that the list is not empty, finding the element to be removed, and then handling one of four cases: the element to be removed is the only element in the list, the element to be removed is the first element in the list, the element to be removed is the last element in the list, or the element to be removed is in the middle of the list. In all cases, the `count` is decremented by one. Unlike the `remove` operation for the array version, the linked version does not require elements to be shifted to close the gap. However, given that the worst case still requires n comparisons to determine that the target element is not in the list, the `remove` operation is still O(n). An implementation of the `remove` operation is shown below.

```
/**
 * Removes the first instance of the specified element from this
 * list and returns a reference to it. Throws an EmptyListException
 * if the list is empty. Throws a NoSuchElementException if the
 * specified element is not found in the list.
 *
 * @param targetElement              the element to be removed from the list
 * @return                           a reference to the removed element
 * @throws EmptyCollectionException  if an empty collection exception occurs
 */
public T remove (T targetElement) throws EmptyCollectionException,
      ElementNotFoundException
{
   if (isEmpty())
      throw new EmptyCollectionException ("List");

   boolean found = false;
   LinearNode<T> previous = null;
   LinearNode<T> current = head;

   while (current != null && !found)
      if (targetElement.equals (current.getElement()))
         found = true;
      else
      {
         previous = current;
         current = current.getNext();
      }

   if (!found)
      throw new ElementNotFoundException ("List");

   if (size() == 1)
      head = tail = null;
   else if (current.equals (head))
      head = current.getNext();
      else if (current.equals (tail))
      {
         tail = previous;
         tail.setNext(null);
      }
      else
         previous.setNext(current.getNext());

      count--;

      return current.getElement();
   }
```

Doubly Linked Lists

Note how much code in this method is devoted to finding the target element and keeping track of a `current` and a `previous` reference. This seems like a missed opportunity to reuse code since we already have a `find` method in the `LinkedList` class. What if this list were *doubly linked*, meaning that each node stores a reference to the next element as well as to the previous element? Would this make the `remove` operation simpler? First, we would need a `DoubleNode` class, as shown in Listing 6.9.

LISTING 6.9

```
/**
 * DoubleNode represents a node in a doubly linked list.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @author Davis
 * @version 1.0, 08/13/08
 */

package jss2;

public class DoubleNode<E>
{
    private DoubleNode<E> next;
    private E element;
    private DoubleNode<E> previous;
```

LISTING 6.9 *continued*

```
/**
 * Creates an empty node.
 */
public DoubleNode()
{
   next = null;
   element = null;
   previous = null;
}

/**
 * Creates a node storing the specified element.
 *
 * @param elem the element to be stored into the new node
 */
public DoubleNode (E elem)
{
   next = null;
   element = elem;
   previous = null;
}

/**
 * Returns the node that follows this one.
 *
 * @return the node that follows the current one
 */
public DoubleNode<E> getNext()
{
   return next;
}

/**
 * Returns the node that precedes this one.
 *
 * @return the node that precedes the current one
 */
public DoubleNode<E> getPrevious()
{
   return previous;
}
```

LISTING 6.9 *continued*

```
   /**
    * Sets the node that follows this one.
    *
    * @param dnode the node to be set as the one to follow the current one
    */
   public void setNext (DoubleNode<E> dnode)
   {
      next = dnode;
   }

   /**
    * Sets the node that precedes this one.
    *
    * @param dnode the node to be set as the one to precede the current one
    */
   public void setPrevious (DoubleNode<E> dnode)
   {
      previous = dnode;
   }

   /**
    * Returns the element stored in this node.
    *
    * @return the element stored in this node
    */
   public E getElement()
   {
      return element;
   }

   /**
    * Sets the element stored in this node.
    *
    * @param elem the element to be stored in this node
    */
   public void setElement (E elem)
   {
      element = elem;
   }
}
```

```
/**
 * Removes and returns the specified element.
 *
 * @param element                     the element to be removed and returned
 *                                    from the list
 * @return                            the element that has been removed from
 *                                    the list
 * @throws ElementNotFoundException   if an element not found exception occurs
 */
public T remove (T element)
{
   T result;
   DoubleNode<T> nodeptr = find (element);

   if (nodeptr == null)
      throw new ElementNotFoundException ("list");

   result = nodeptr.getElement();

   // check to see if front or rear
   if (nodeptr == front)
      result = this.removeFirst();
   else if (nodeptr == rear)
        result = this.removeLast();
        else
   {
        nodeptr.getNext().setPrevious(nodeptr.getPrevious());
               nodeptr.getPrevious().setNext(nodeptr.getNext());
      count—;
   }
  return result;
}
```

The `remove` operation can now be implemented much more elegantly using a doubly linked list. Note that we can now use the `find` operation to locate the target, and we no longer need to keep track of a `previous` reference. In this example, we also use the `removeFirst` and `removeLast` operations to handle the special cases associated with removing either the first or last element.

The `iterator` Operation

The `iterator` method simply returns a new `LinkedIterator` object:

```
/**
 * Returns an iterator for the elements currently in this list.
 *
 * @return an iterator over the elements of this list
 */
public Iterator<T> iterator()
{
   return new LinkedIterator<T>(head, count);
}
```

Like the ArrayIterator<T> class discussed earlier, the LinkedIterator<T> class is written so that it can be used with multiple collections. It stores the contents of the linked list and the count of elements, as shown in Listing 6.10.

LISTING 6.10

```
/**
 * LinkedIterator represents an iterator for a linked list of linear nodes.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 08/13/08
 */

package jss2;
import jss2.exceptions.*;
import java.util.*;

public class LinkedIterator<T> implements Iterator<T>
{
  private int count; // the number of elements in the collection
  private LinearNode<T> current; // the current position

  /**
   * Sets up this iterator using the specified items.
   *
   * @param collection  the collection the iterator will move over
   * @param size        the integer size of the collection
   */
  public LinkedIterator (LinearNode<T> collection, int size)
  {
    current = collection;
    count = size;
  }
```

LISTING 6.10 *continued*

```
    /**
     * Returns true if this iterator has at least one more element
     * to deliver in the iteration.
     *
     * @return true if this iterator has a least one more element to deliver
     *          in the iteration
     */
    public boolean hasNext()
    {
      return (current!= null);
    }

    /**
     * Returns the next element in the iteration. If there are no
     * more elements in this iteration, a NoSuchElementException is
     * thrown.
     *
     * @return                        the next element in the iteration
     * @throws NoSuchElementException  if a no such element exception occurs
     */
    public T next()
    {
      if (! hasNext())
          throw new NoSuchElementException();
      T result = current.getElement();
      current = current.getNext();
      return result;
    }

    /**
     * The remove operation is not supported.
     *
     * @throws UnsupportedOperationException  if an unsupported operation
     *                                        exception occurs
     */
    public void remove() throws UnsupportedOperationException
    {
      throw new UnsupportedOperationException();
    }
}
```

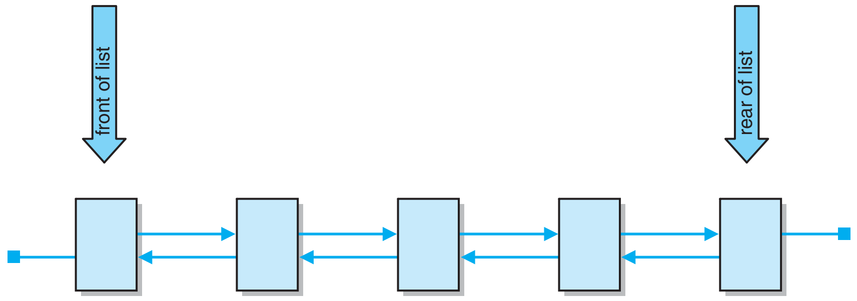


FIGURE 6.13 A doubly linked list

The `LinkedIterator` constructor sets up a reference that is designed to move across the list of elements in response to calls to the `next` method. The iteration is complete when `current` becomes null, which is the condition returned by the `hasNext` method. As in the case of the `ArrayIterator<T>` class from earlier in this chapter, the `remove` method is left unsupported.

Figure 6.13 illustrates the structure of a doubly linked list. The implementations of the other operations for both linked and doubly linked lists are left as exercises.

6.6 Lists in the Java Collections API

The Java Collections API provides three separate implementations for an indexed list: `Vector`, `ArrayList` and `LinkedList`. All three of these classes extend the abstract class `java.util.AbstractList`, which implements the `java.util.List` interface. These are part of the Java class library and are distinct from the interfaces and classes we have discussed so far in this chapter. The `java.util.AbstractList` class is an extension of the `java.util.AbstractCollection` class, which implements the `java.util.Collection` interface. All of the list implementations provided in the Java Collections API framework are indexed lists, even though the class names do not identify them as such.

All three of these list implementations share some common attributes. First, they all implement the `java.util.List` interface. The precise details of the abstract methods in this interface are available online but suffice to say, there are many more methods than those in our simple implementation from earlier in the chapter. Keep in mind that the Java language is a very rich, industrial-strength language, not necessarily designed for education. Thus, as we have discussed previously, many of the collections within the Java Collections API do not precisely conform to the conceptual definition of their respective collections. They often provide a larger variety of operations than the conceptual collection prescribes.

In addition to the `Collection` and `List` interfaces, all three varieties of lists implement the `Cloneable` and the `Serializable` interfaces as well.

`Cloneable`

Implementing the `Cloneable` interface does not necessarily require any additional methods. The interface does not contain a clone method or any other abstract methods. This interface simply indicates to the Object.clone() method that this class allows the method to make a field-for-field copy of its instances. Typically, classes that implement the `Cloneable` interface override the `Object.clone()` method with a public method of their own.

`Serializable`

Although serializability in general can provide a number of advantages, the typical use in our context is that a class that implements the `Serializable` interface can be decomposed into a byte stream and then written to memory or to disk using the `ObjectOutputStream` class of the `java.io` package. This byte stream can then be reconstructed into the original object using the `ObjectInputStream` class, also in the `java.io` package.

Why is this important? Consider that when you create a class in Java source code (e.g., `someclass.java`), you are not creating data but simply creating a blueprint or structure for data. To this point, our data has typically been hard coded into our programs, input by the user as the program executes, or read from a file as a stream of text and then parsed into the appropriate fields of objects. This means that while the classes that we have been creating have been persistent, the instances of those classes have not. The instances simply cease to exist when the program terminates. Creating classes that implement the `Serializable` interface allows instances of those classes to be written to memory or to disk and then reconstructed at a later date without the developer having to rebuild them from text.

Like the `Cloneable` interface, there are not any methods associated with the `Serializable` interface. This interface is simply a marker notifying the run-time environment that objects of this class are to be serialized.

`RandomAccess`

Both the `Vector` class and the `ArrayList` class also implement the `RandomAccess` interface. Like both the `Cloneable` interface and the `Serializable` interface, there are not any methods associated with the `RandomAccess` interface. This interface is simply a marker indicating that these classes provide fast random access. Both `Vector` and `ArrayList` are array-based implementations and as we have seen with our own implementations earlier in this chapter, this gives us O(1) access to elements in the list.

KEY CONCEPT
The Java Collections API contains three implementations of an indexed list.

Keep in mind that operations other than simply accessing elements may be more complex. In fact we know that insertion and removal of elements will be O(n) based on the need to shift elements.

`Java.util.Vector`

The `Vector` implementation of an indexed list is an array-based implementation. Thus, many of the issues discussed in the array implementations of stacks, queues, unordered lists, and ordered lists apply here as well. For example, using an array implementation of a list, an `add` operation that specifies an index in the middle of the list will require all of the elements above that position in the list to be shifted one position higher in the list. Likewise, a `remove` operation that removes an element from the middle of the list will require all of the elements above that position in the list to be shifted one position lower in the list.

The `Vector` class implements the `Serializable`, `Cloneable`, `Iterable<E>`, `Collection<E>`, `Deque<E>`, `List<E>`, and `Queue<E>` interfaces. Figure 6.14 illustrates the public and protected methods available for the `Vector` class. The `Vector` class also inherits quite a number of additional methods from its superclasses.

`Java.util.ArrayList`

The `ArrayList` implementation of an indexed list is, as its name implies, an array-based implementation. Thus, just as with the implementation of the `Vector` class, many of the issues discussed in the array implementations of stacks, queues, unordered lists, and ordered lists apply here as well. For example, using an array implementation of a list, an `add` operation that specifies an index in the middle of the list will require all of the elements above that position in the list to be shifted one position higher in the list. Likewise, a `remove` operation that removes an element from the middle of the list will require all of the elements above that position in the list to be shifted one position lower in the list.

Like the `Vector` class, the `ArrayList` implementation is resizable, meaning that if adding the next element would overflow the `ArrayList`, the underlying array is automatically resized. To manage the size of the array, both classes contain two additional operations: `ensureCapacity` increases the size of the array to the specified size if it is not already that large or larger, and `trimToSize` trims the array to the actual current size of the list.

The `ArrayList` implementation is very similar to the implementation of a `Vector`. However, the `Vector` operations are synchronized, and `ArrayList` operations are not. Figure 6.15 illustrates the public and protected methods available for the `ArrayList` class. The `ArrayList` class also inherits quite a number of additional methods from its superclasses. The `ArrayList` class implements the `Serializable`, `Coneable`, `Iterable<E>`, `Collection<E>`, `List<E>`, and `RandomAccess` interfaces.

Method Signature	Return Type	Description
`add(E e)`	boolean	Appends the specified element to the end of this Vectorvector.
`add(int index, E element)`	void	Inserts the specified element at the specified position in this vector.
`addAll(Collection <? extends E > c)`	boolean	Appends all of the elements in the specified collection to the end of this vector, in the order that they are returned by the specified collection's iterator.
`addAll(int index, Collection<? extends E > c)`	boolean	Inserts all of the elements in the specified collection into this vector at the specified position.
`addElement(E obj)`	void	Adds the specified component to the end of this vector, increasing its size by one.
`capacity()`	int	Returns the current capacity of this vector.
`clear()`	void	Removes all of the elements from this vector.
`clone()`	Object	Returns a clone of this vector.
`contains(Object o)`	boolean	Returns true if this vector contains the specified element.
`containsAll (Collection<?> c)`	boolean	Returns true if this vector contains all of the elements in the specified collection.
`copyInto(Object[] anArray)`	void	Copies the components of this vector into the specified array.
`elementAt(int index)`	E	Returns the component at the specified index.
`elements()`	Enumeration<E>	Returns an enumeration of the components of this vector.
`ensureCapacity(int minCapacity)`	void	Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.
`equals(Object o)`	boolean	Compares the specified object with this vector for equality.
`firstElement()`	E	Returns the first component (the item at index 0) of this vector.
`get(int index)`	E	Returns the element at the specified position in this vector.
`hashCode()`	int	Returns the hash code value for this vector.
`indexOf(Object o)`	int	Returns the index of the first occurrence of the specified element in this vector, or -1 if this vector does not contain the element.

FIGURE 6.14 Public and protected methods of the `java.util.Vector` class

Method Signature	Return Type	Description
`indexOf(Object o, int index)`	int	Returns the index of the first occurrence of the specified element in this vector, searching forwards from index, or returns -1 if the element is not found.
`insertElementAt (E obj, int index)`	void	Inserts the specified object as a component in this vector at the specified index.
`isEmpty()`	boolean	Tests if this vector has no components.
`lastElement()`	E	Returns the last component of the vector.
`lastIndexOf(Object o)`	int	Returns the index of the last occurrence of the specified element in this vector, or −1 if this vector does not contain the element.
`lastIndexOf(Object o, int index)`	int	Returns the index of the last occurrence of the specified element in this vector, searching backwards from index, or returns −1 if the element is not found.
`remove(int index)`	E	Removes the element at the specified position in this vector.
`remove(Object o)`	boolean	Removes the first occurrence of the specified element in this vector. If the vector does not contain the element, it is unchanged.
`removeAll (Collection<?> c)`	boolean	Removes from this vector all of its elements that are contained in the specified collection.
`removeAllElements()`	void	Removes all components from this vector and sets its size to zero.
`removeElement(Object obj)`	boolean	Removes the first (lowest-indexed) occurrence of the argument from this vector.
`removeElementAt(int index)`	void	Deletes the component at the specified index.
`removeRange(int fromIndex, int toIndex)`	void	Removes from this list all of the elements whose index is between `fromIndex`, inclusive and `toIndex`, exclusive.
`retainAll (Collection<?> c)`	boolean	Retains only the elements in this vector that are contained in the specified collection.
`set(int index, E element)`	E	Replaces the element at the specified position in this vector with the specified element.
`setElementAt(E obj, int index)`	void	Sets the component at the specified index of this vector to be the specified object.
`setSize(int newSize)`	void	Sets the size of this vector.
`size()`	int	Returns the number of components in this vector.

FIGURE 6.14 *Continued*

Method Signature	Return Type	Description
`subList(int fromIndex, int toIndex)`	List<E>	Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive.
`toArray()`	Object[]	Returns an array containing all of the elements in this vector in the correct order.
`toArray(T[] a)`	<T> T[]	Returns an array containing all of the elements in this vector in the correct order; the runtime type of the returned array is that of the specified array.
`toString()`	String	Returns a string representation of this vector, containing the String representation of each element.
`trimToSize()`	void	Trims the capacity of this vector to be the vector's current size.

FIGURE 6.14 *Continued*

Like arrays and the `Vector` implementation, one advantage of the ArrayList implementation is the ability to access any element in the list in equal time. However, the penalty for that access is the added cost of shifting remaining elements either as part of an insertion into the list or a deletion from the list.

Java.util.LinkedList

The `LinkedList` implementation of an indexed list is, as the name implies, a linked implementation. Thus, many of the issues discussed in the linked implementations of stacks, queues, unordered lists, and ordered lists apply here as well. For example, using a linked implementation of a list requires us to traverse the list to access a particular element instead of going directly to it by index. For this reason, the `LinkedList` class does not implement the `RandomAccess` interface because it does not provide constant or near constant time access to its elements.

In addition to the interfaces discussed previously that all of the Java Collection API implementations of a list implement, the `LinkedList` class also implements both the `Queue` interface and the `Deque` interface (pronounced deck and meaning double-ended queue). As we discussed earlier, this is an artifact of the fact that Java is an industrial language and was not necessarily designed for education. As we will see, this use of the `LinkedList` class to implement a queue creates a collection that will allow actions inappropriate for a queue.

The `LinkedList` class implements the `Serializable`, `Cloneable`, `Iterable<E>`, `Collection<E>`, `Deque<E>`, `List<E>`, and `Queue<E>` interfaces. Figure 6.16 illustrates the public and protected methods available for the `LinkedList` class. The `LinkedList` class also inherits quite a number of additional methods from its superclasses.

Method Signature	Return Type	Description
`add(E e)`	boolean	Appends the specified element to the end of this list.
`add(int index, E element)`	void	Inserts the specified element at the specified position in this list.
`addAll(Collection<? extends E > c)`	boolean	Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
`addAll(int index, Collection<? extends E > c)`	boolean	Inserts all of the elements in the specified collection into this list, starting at the specified position.
`clear()`	void	Removes all of the elements from this list.
`clone()`	Object	Returns a shallow copy of this `ArrayList` instance.
`contains(Object o)`	boolean	Returns true if this list contains the specified element.
`ensureCapacity(int minCapacity)`	void	Increases the capacity of this `ArrayList` instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
`get(int index)`	E	Returns the element at the specified position in this list.
`indexOf(Object o)`	int	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
`isEmpty()`	boolean	Returns true if this list contains no elements.
`lastIndexOf(Object o)`	int	Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
`remove(int index)`	E	Removes the element at the specified position in this list.
`remove(Object o)`	boolean	Removes the first occurrence of the specified element from this list, if it is present.
`removeRange(int fromIndex, int toIndex)`	void	Removes from this list all of the elements whose index is between `fromIndex`, inclusive, and `toIndex`, exclusive.
`set(int index, E element)`	E	Replaces the element at the specified position in this list with the specified element.
`size()`	int	Returns the number of elements in this list.
`toArray()`	Object[]	Returns an array containing all of the elements in this list in proper sequence (from first to last element).
`toArray(T[] a)`	`<T> T[]`	Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.
`trimToSize()`	void	Trims the capacity of this `ArrayList` instance to be the list's current size.

FIGURE 6.15 Public and protected methods of the `java.util.ArrayList` class

Method Signature	Return Type	Description
`add(E e)`	boolean	Appends the specified element to the end of this list.
`add(int index, E element)`	void	Inserts the specified element at the specified position in this list.
`addAll(Collection<? extends E > c)`	boolean	Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
`addAll(int index, Collection<? extends E > c)`	boolean	Inserts all of the elements in the specified collection into this list, starting at the specified position.
`addFirst(E e)`	void	Inserts the specified element at the beginning of the list.
`addLast(E e)`	void	Appends the specified element at the end of the list.
`clear()`	void	Removes all of the elements from this list.
`clone()`	Object	Returns a shallow copy of this `ArrayList` instance.
`contains(Object o)`	boolean	Returns true if this list contains the specified element.
`descendingIterator()`	Iterator<E>	Returns an iterator over the elements in this deque in reverse sequential order.
`element()`	E	Retrieves, but does not remove, the head (first element) of this list.
`get(int index)`	E	Returns the element at the specified position in this list.
`getFirst()`	E	Returns the first element in this list.
`getLast()`	E	Returns the last element in this list.
`indexOf(Object o)`	int	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
`lastIndexOf(Object o)`	int	Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
`listIterator(int index)`	ListIterator<E>	Returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list.
`offer(E e)`	boolean	Adds the specified element as the tail (last element) of the list.
`offerFirst(E e)`	boolean	Inserts the specified element at the front of this list.
`offerLast(E e)`	boolean	Inserts the specified element at the end of this list.
`peek()`	E	Retrieves, but does not remove, the first element in the list.

FIGURE 6.16 Public and protected methods of the `java.util.LinkedList` class

Method Signature	Return Type	Description
`peekFirst()`	E	Retrieves, but does not remove, the first element in the list or returns null if the list is empty.
`peekLast()`	E	Retrieves, but does not remove, the last element in the list or returns null if the list is empty.
`poll()`	E	Retrieves and removes the head (first element) of this list.
`pollFirst()`	E	Retrieves and removes the first element of this list or returns null if the list is empty.
`pollLast()`	E	Retrieves and removes the last element of this list or returns null if the list is empty.
`pop()`	E	Pops an element from the stack represented by this list.
`push(E e)`	void	Pushes an element onto the stack represented by this list.
`remove()`	E	Retrieves and removes the head (first element) of this list.
`remove(int index)`	E	Removes the element at the specified position in this list.
`remove(Object o)`	boolean	Removes the first occurrence of the specified element from this list, if it is present.
`removeFirst()`	E	Removes and returns the first element from this list.
`removeFirstOccurrence (Object o)`	boolean	Removes the first occurrence of the specified element from this list.
`removeLast()`	E	Removes and returns the last element from this list.
`removeLastOccurrence (Object o)`	`boolean`	Removes the last occurrence of the specified element from this list.
`set(int index, E element)`	E	Replaces the element at the specified position in this list with the specified element.
`size()`	int	Returns the number of elements in this list.
`toArray()`	Object[]	Returns an array containing all of the elements in this list in proper sequence (from first to last element).
`toArray(T[] a)`	<T> T[]	Returns an array containing all of the elements in this list in proper sequence (from first to last element). The runtime type of the returned array is that of the specified array.

FIGURE 6.16 *Continued*

Summary of Key Concepts

- List collections can be categorized as ordered, unordered, and indexed.
- The elements of an ordered list have an inherent relationship defining their order.
- The elements of an unordered list are kept in whatever order the client chooses.
- An indexed list maintains a contiguous numeric index range for its elements.
- Many common operations can be defined for all list types. The differences between them stem from how elements are added.
- An iterator is an object that provides a means to iterate over a collection.
- Interfaces can be used to derive other interfaces. The child interface contains all abstract methods of the parent.
- An interface name can be used to declare an object reference variable. An interface reference can refer to any object of any class that implements the interface.
- Interfaces allow us to make polymorphic references in which the method that is invoked is based on the particular object being referenced at the time.
- An ordered list is a convenient collection to use when creating a tournament schedule.
- The Josephus problem is a classic computing problem that is appropriately solved with indexed lists.
- Only `Comparable` objects can be stored in an ordered list.
- The Java Collections API contains three implementations of an indexed list.

Self-Review Questions

SR 6.1 What is the difference between an indexed list, an ordered list, and an unordered list?

SR 6.2 What are the basic methods of accessing an indexed list?

SR 6.3 What are the additional operations required of implementations that are part of the Java Collections API framework?

SR 6.4 What are the trade-offs in space complexity between an `ArrayList` and a `LinkedList`?

SR 6.5 What are the trade-offs in time complexity between an `ArrayList` and a `LinkedList`?

SR 6.6 What is the time complexity of the `contains` operation and the `find` operation for both implementations?

SR 6.7 What effect would it have if the `LinkedList` implementation were to use a singly linked list instead of a doubly linked list?

SR 6.8 Why is the time to increase the capacity of the array on an `add` operation considered negligible for the `ArrayList` implementation?

SR 6.9 What is an iterator and why is it useful for ADTs?

SR 6.10 Why was an iterator not appropriate for stacks and queues but is appropriate for lists?

SR 6.11 Why is a circular array implementation not as attractive as an implementation of a list as it was for a queue?

Exercises

EX 6.1 Hand trace an ordered list X through the following operations:

```
X.add(new Integer(4));
X.add(new Integer(7));
Object Y = X.first();
X.add(new Integer(3));
X.add(new Integer(2));
X.add(new Integer(5));
Object Y = X.removeLast();
Object Y = X.remove(new Integer(7));
X.add(new Integer(9));
```

EX 6.2 Given the resulting list X from Exercise 6.1, what would be the result of each of the following?

```
a.  X.last();
b.  z = X.contains(new Integer(3));
    X.first();
c.  Y = X.remove(new Integer(2));
    X.first();
```

EX 6.3 What would be the time complexity of the `size` operation for each of the implementations if there were not a `count` variable?

EX 6.4 In the array implementation, under what circumstances could the `head` and `tail` references be equal?

EX 6.5 In the linked implementation, under what circumstances could the `head` and `tail` references be equal?

EX 6.6 If there were not a `count` variable in the array implementation, how could you determine whether or not the list was empty?

EX 6.7 If there were not a `count` variable in the array implementation, how could you determine whether or not the list was full?

Programming Projects

PP 6.1 Implement a stack using a `LinkedList`.

PP 6.2 Implement a stack using an `ArrayList`.

PP 6.3 Implement a queue using a `LinkedList`.

PP 6.4 Implement a queue using an `ArrayList`.

PP 6.5 Implement the Josephus problem using a queue, and compare the performance of that algorithm to the `ArrayList` implementation from this chapter.

PP 6.6 Implement an `OrderedList` using a `LinkedList`.

PP 6.7 Implement an `OrderedList` using an `ArrayList`.

PP 6.8 Complete the implementation of the `ArrayList` class.

PP 6.9 Complete the implementation of the `ArrayOrderedList` class.

PP 6.10 Complete the implementation of the `ArrayUnorderedList` class.

PP 6.11 Write an implementation of the `LinkedList` class.

PP 6.12 Write an implementation of the `LinkedOrderedList` class.

PP 6.13 Write an implementation of the `LinkedUnorderedList` class.

PP 6.14 Create an implementation of a doubly linked `DoubleOrderedList` class. You will need to create a `DoubleNode` class, a `DoubleList` class, and a `DoubleIterator` class.

PP 6.15 Create a graphical application that provides a button for `add` and `remove` from an ordered list, a text field to accept a string as input for `add`, and a text area to display the contents of the list after each operation.

PP 6.16 Create a graphical application that provides a button for `addToFront`, `addToRear`, `addAfter`, and `remove` from an unordered list. Your application must provide a text field to accept a string as input for any of the `add` operations. The user should be able to select the element to be added after, and select the element to be removed.

PP 6.17 Modify the `TournamentMaker` program from this chapter so that the user may select some number of teams to receive a bye in the first round. For example, in a ten team league, the user may specify that the top two teams would automatically advance to the second round to play the winner of the contest between teams 6 and 7, and 5 and 8, respectively.

Answers to Self-Review Questions

SRA 6.1 An indexed list is a collection of objects with no inherent order that are ordered by index value. An ordered list is a collection of objects ordered by value. An unordered list is a collection of objects with no inherent order.

SRA 6.2 Access to the list is accomplished in one of three ways: by accessing a particular index position in the list, by accessing the ends of the list, or by accessing an object in the list by value.

SRA 6.3 All Java Collections API framework classes implement the `Collections` interface, the `Serializable` interface, and the `Cloneable` interface.

SRA 6.4 The linked implementation requires more space per object to be inserted in the list simply because of the space allocated for the references. Keep in mind that the `LinkedList` class is actually a doubly linked list, thus requiring twice as much space for references. The `ArrayList` class is more efficient at managing space than the array-based implementations we have discussed previously. This is due to the fact that `ArrayList` collections are resizable, and thus can dynamically allocate space as needed. Therefore, there need not be a large amount of wasted space allocated all at once. Rather, the list can grow as needed.

SRA 6.5 The major difference between the two is access to a particular index position of the list. The `ArrayList` implementation can access any element of the list in equal time if the index value is known. The `LinkedList` implementation requires the list to be traversed from one end or the other to reach a particular index position.

SRA 6.6 The `contains` and `find` operations for both implementations will be O(n) because they are simply linear searches.

SRA 6.7 This would change the time complexity for the `addToRear` and `removeLast` operations because they would now require traversal of the list.

SRA 6.8 Averaged over the total number of insertions into the list, the time to enlarge the array has little effect on the total time.

SRA 6.9 An iterator is an object that provides a means of stepping through the elements of a collection one at a time.

SRA 6.10 Conceptually, stacks and queues should not allow access to elements in the middle of the list. Stacks only allow access to the top and queues only allow the addition of elements on one end and removal of elements on the other. An iterator would violate the conceptual definition of these collections. Lists, on the other hand, allow access throughout and are perfectly suited for an iterator.

SRA 6.11 The circular array implementation of a queue improved the efficiency of the `dequeue` operation from O(n) to O(1) because it eliminated the need to shift elements in the array. That is not the case for a list because we can add or remove elements anywhere in the list, not just at the front or the rear.

7 Recursion

Recursion is a powerful programming technique that provides elegant solutions to certain problems. It is particularly helpful in the implementation of various data structures and in the process of searching and sorting data. This chapter provides an introduction to recursive processing. It contains an explanation of the basic concepts underlying recursion and then explores the use of recursion in programming.

CHAPTER OBJECTIVES

- Explain the underlying concepts of recursion
- Examine recursive methods and unravel their processing steps
- Define infinite recursion and discuss ways to avoid it
- Explain when recursion should and should not be used
- Demonstrate the use of recursion to solve problems

7.1 Recursive Thinking

KEY CONCEPT
Recursion is a programming technique in which a method calls itself. A key to being able to program recursively is to be able to think recursively.

We know that one method can call another method to help it accomplish its goal. Similarly, a method can also call itself to help accomplish its goal. *Recursion* is a programming technique in which a method calls itself to fulfill its overall purpose.

Before we get into the details of how we use recursion in a program, we need to explore the general concept of recursion first. The ability to think recursively is essential to being able to use recursion as a programming technique.

In general, recursion is the process of defining something in terms of itself. For example, consider the following definition of the word *decoration*:

decoration: n. any ornament or adornment used to decorate something

The word *decorate* is used to define the word *decoration*. You may recall your grade-school teacher telling you to avoid such recursive definitions when explaining the meaning of a word. However, in many situations, recursion is an appropriate way to express an idea or definition. For example, suppose we want to formally define a list of one or more numbers, separated by commas. Such a list can be defined recursively either as a number or as a number followed by a comma followed by a list. This definition can be expressed as follows:

A list is a:	`number`
or a:	`number  comma  list`

This recursive definition of a list defines each of the following lists of numbers:

```
24, 88, 40, 37
96, 43
14, 64, 21, 69, 32, 93, 47, 81, 28, 45, 81, 52, 69
70
```

No matter how long a list is, the recursive definition describes it. A list of one element, such as in the last example, is defined completely by the first (nonrecursive) part of the definition. For any list longer than one element, the recursive part of the definition (the part that refers to itself) is used as many times as necessary, until the last element is reached. The last element in the list is always defined by the nonrecursive part of this definition. Figure 7.1 shows how one particular list of numbers corresponds to the recursive definition of *list*.

Infinite Recursion

Note that this definition of a list contains one option that is recursive, and one option that is not. The part of the definition that is not recursive is called the *base case*. If all options had a recursive component, then the recursion would

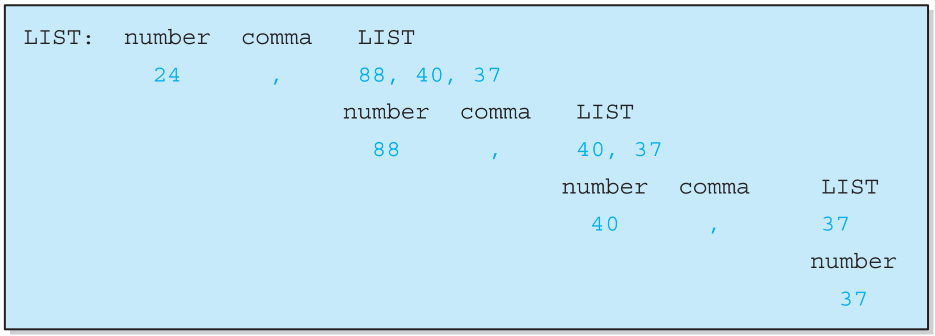


FIGURE 7.1 Tracing the recursive definition of a list

never end. For example, if the definition of a list were simply "a number followed by a comma followed by a list," then no list could ever end. This problem is called *infinite recursion*. It is similar to an infinite loop, except that the "loop" occurs in the definition itself.

KEY CONCEPT
Any recursive definition must have a nonrecursive part, called the base case, which permits the recursion to eventually end.

As in the infinite loop problem, a programmer must be careful to design algorithms so that they avoid infinite recursion. Any recursive definition must have a base case that does not result in a recursive option. The *base case* of the list definition is a single number that is not followed by anything. In other words, when the last number in the list is reached, the base case option terminates the recursive path.

Recursion in Math

Let's look at an example of recursion in mathematics. The value referred to as N! (which is pronounced *N factorial*) is defined for any positive integer N as the product of all integers between 1 and N inclusive. Therefore:

```
3! = 3*2*1 = 6
```

and

```
5! = 5*4*3*2*1 = 120.
```

Mathematical formulas are often expressed recursively. The definition of N! can be expressed recursively as:

```
1! = 1
N! = N * (N-1)! for N > 1
```

KEY CONCEPT
Mathematical problems and formulas are often expressed recursively.

The base case of this definition is 1!, which is defined to be 1. All other values of N! (for N > 1) are defined recursively as N times the value (N–1)!. The recursion is that the factorial function is defined in terms of the factorial function.

Using this definition, 50! is equal to 50 * 49!. And 49! is equal to 49 * 48!. And 48! is equal to 48 * 47!. This process continues until we get to the base case of 1. Because N! is defined only for positive integers, this definition is complete and will always conclude with the base case.

The next section describes how recursion is accomplished in programs.

7.2 Recursive Programming

Let's use a simple mathematical operation to demonstrate the concepts of recursive programming. Consider the process of summing the values between 1 and N inclusive, where N is any positive integer. The sum of the values from 1 to N can be expressed as N plus the sum of the values from 1 to N–1. That sum can be expressed similarly, as shown in Figure 7.2.

KEY CONCEPT
Each recursive call to a method creates new local variables and parameters.

For example, the sum of the values between 1 and 20 is equal to 20 plus the sum of the values between 1 and 19. Continuing this approach, the sum of the values between 1 and 19 is equal to 19 plus the sum of the values between 1 and 18. This may sound like a strange way to think about this problem, but it is a straightforward example that can be used to demonstrate how recursion is programmed.

In Java, as in many other programming languages, a method can call itself. Each call to the method creates a new environment in which to work. That is, all

$$\begin{aligned}\sum_{i=1}^{N} \boldsymbol{i} &= N + \sum_{i=1}^{N-1} \boldsymbol{i} = N + N-1 + \sum_{i=1}^{N-2} \boldsymbol{i} \\ &= N + N-1 + N-2 + \sum_{i=1}^{N-3} \boldsymbol{i} \\ &= N + N-1 + N-2 + \ldots + 2 + 1\end{aligned}$$

FIGURE 7.2 The sum of the numbers 1 through N, defined recursively

local variables and parameters are newly defined with their own unique data space every time the method is called. Each parameter is given an initial value based on the new call. Each time a method terminates, processing returns to the method that called it (which may be an earlier invocation of the same method). These rules are no different from those governing any "regular" method invocation.

A recursive solution to the summation problem is defined by the following recursive method called sum:

```
// This method returns the sum of 1 to num
public int sum (int num)
{
   int result;
   if (num == 1)
      result = 1;
   else
      result = num + sum (num-1);
   return result;
}
```

Note that this method essentially embodies our recursive definition that the sum of the numbers between 1 and N is equal to N plus the sum of the numbers between 1 and N–1. The sum method is recursive because sum calls itself. The parameter passed to sum is decremented each time sum is called, until it reaches the base case of 1. Recursive methods usually contain an if-else statement, with one of the branches representing the base case.

KEY CONCEPT
A careful trace of recursive processing can provide insight into the way it is used to solve a problem.

Suppose the main method calls sum, passing it an initial value of 1, which is stored in the parameter num. Because num is equal to 1, the result of 1 is returned to main, and no recursion occurs.

Now let's trace the execution of the sum method when it is passed an initial value of 2. Because num does not equal 1, sum is called again with an argument of num-1, or 1. This is a new call to the method sum, with a new parameter num and a new local variable result. Because this num is equal to 1 in this invocation, the result of 1 is returned without further recursive calls. Control returns to the first version of sum that was invoked. The return value of 1 is added to the initial value of num in that call to sum, which is 2. Therefore, result is assigned the value 3, which is returned to the main method. The method called from main correctly calculates the sum of the integers from 1 to 2, and returns the result of 3.

The base case in the summation example is when N equals 1, at which point no further recursive calls are made. The recursion begins to fold back into the earlier versions of the sum method, returning the appropriate value each time. Each return value contributes to the computation of the sum at the higher level. Without the base case, infinite recursion would result. Each call to a method requires additional

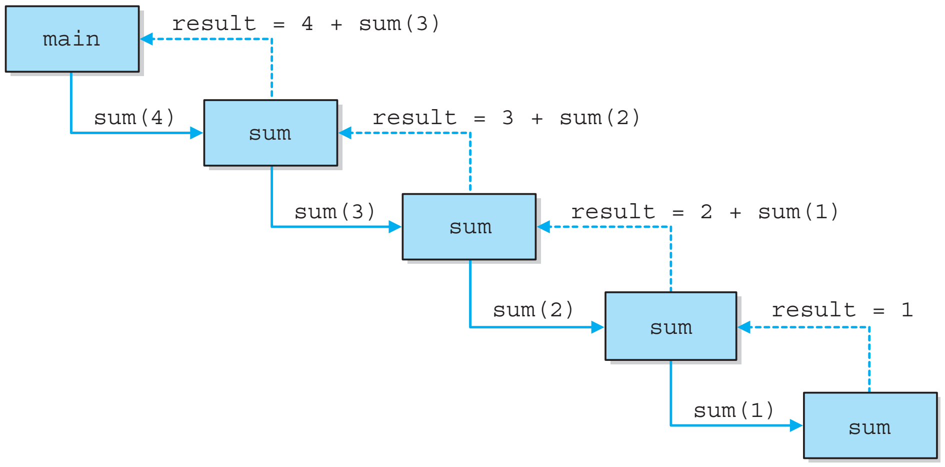


FIGURE 7.3 Recursive calls to the `sum` method

memory space; therefore, infinite recursion often results in a run-time error indicating that memory has been exhausted.

Trace the `sum` function with different initial values of `num` until this processing becomes familiar. Figure 7.3 illustrates the recursive calls when `main` invokes `sum` to determine the sum of the integers from 1 to 4. Each box represents a copy of the method as it is invoked, indicating the allocation of space to store the formal parameters and any local variables. Invocations are shown as solid lines, and returns are shown as dotted lines. The return value `result` is shown at each step. The recursive path is followed completely until the base case is reached; then the calls begin to return their result up through the chain.

Recursion versus Iteration

Of course, there is a nonrecursive solution to the summation problem we just explored. One way to compute the sum of the numbers between 1 and `num` inclusive in an iterative manner is as follows:

```
sum = 0;
for (int number = 1; number <= num; number++)
    sum += number;
```

This solution is certainly more straightforward than the recursive version. We used the summation problem to demonstrate recursion because it is a simple problem to understand, not because you would use recursion to solve it under normal conditions. Recursion has the overhead of multiple method invocations and, in this case, presents a more complicated solution than its iterative counterpart.

A programmer must learn when to use recursion and when not to use it. Determining which approach is best is another important software engineering decision that depends on the problem being solved. All problems can be solved in an iterative manner, but in some cases the iterative version is much more complicated. Recursion, for some problems, allows us to create relatively short, elegant programs.

KEY CONCEPT

Recursion is the most elegant and appropriate way to solve some problems, but for others it is less intuitive than an iterative solution.

Direct versus Indirect Recursion

Direct recursion occurs when a method invokes itself, such as when `sum` calls `sum`. *Indirect recursion* occurs when a method invokes another method, eventually resulting in the original method being invoked again. For example, if method `m1` invokes method `m2`, and `m2` invokes method `m1`, we can say that `m1` is indirectly recursive. The amount of indirection could be several levels deep, as when `m1` invokes `m2`, which invokes `m3`, which invokes `m4`, which invokes `m1`. Figure 7.4 depicts a situation with indirect recursion. Method invocations are shown with solid lines, and returns are shown with dotted lines. The entire invocation path is followed, and then the recursion unravels following the return path.

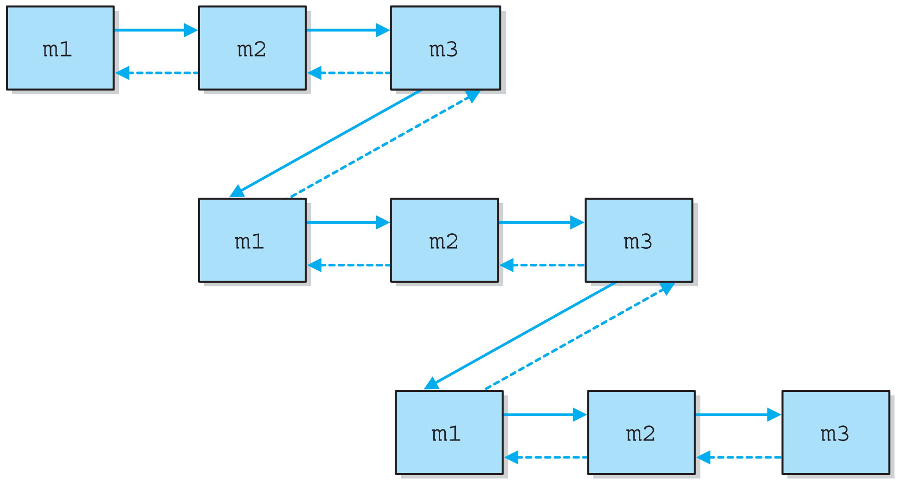


FIGURE 7.4 Indirect recursion

Indirect recursion requires all of the same attention to base cases that direct recursion requires. Furthermore, indirect recursion can be more difficult to trace because of the intervening method calls. Therefore, extra care is warranted when designing or evaluating indirectly recursive methods. Ensure that the indirection is truly necessary and clearly explained in documentation.

7.3 Using Recursion

The following sections describe problems that we then solve using a recursive technique. For each one, we examine exactly how recursion plays a role in the solution and how a base case is used to terminate the recursion. As you explore these examples, consider how complicated a nonrecursive solution for each problem would be.

Traversing a Maze

As we discussed in Chapter 4, solving a maze involves a great deal of trial and error: following a path, backtracking when you cannot go farther, and trying other, untried options. Such activities often are handled nicely using recursion. In Chapter 4, we solved this problem iteratively using a stack to keep track of our potential moves. However, we can also solve this problem recursively by using the run-time stack to keep track of our progress. The program shown in Listing 7.1 creates a `Maze` object and attempts to traverse it.

The `Maze` class, shown in Listing 7.2, uses a two-dimensional array of integers to represent the maze. The goal is to move from the top-left corner (the entry point) to the bottom-right corner (the exit point). Initially, a 1 indicates a clear path, and a 0 indicates a blocked path. As the maze is solved, these array elements are changed to other values to indicate attempted paths and, ultimately, a successful path through the maze if one exists. Figure 7.5 shows the UML illustration of this solution.

The only valid moves through the maze are in the four primary directions: down, right, up, and left. No diagonal moves are allowed. In this example, the

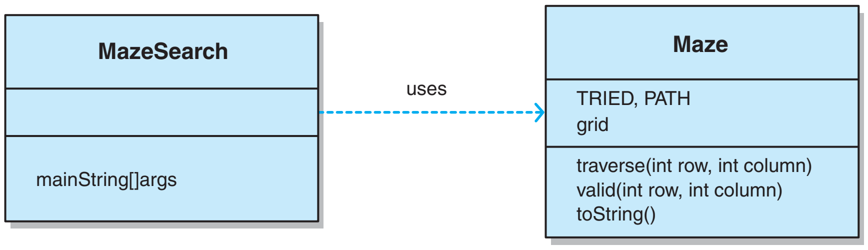


FIGURE 7.5 UML description of the Maze and MazeSearch classes

LISTING 7.1

```java
/**
 * MazeSearch demonstrates recursion.
 *
 * @author Dr. Chase
 * @author Dr. Lewis
 * @version 1.0, 8/18/08
 */

public class MazeSearch
{
   /**
    * Creates a new maze, prints its original form, attempts to
    * solve it, and prints out its final form.
    */
   public static void main (String[] args)
   {
      Maze labyrinth = new Maze();
      System.out.println (labyrinth);

     if (labyrinth.traverse (0, 0))
        System.out.println ("The maze was successfully traversed!");
     else
        System.out.println ("There is no possible path.");

     System.out.println (labyrinth);
   }
}
```

maze is 8 rows by 13 columns, although the code is designed to handle a maze of any size.

Let's think this through recursively. The maze can be traversed successfully if it can be traversed successfully from position (0, 0). Therefore, the maze can be traversed successfully if it can be traversed successfully from any positions adjacent to (0, 0), namely position (1, 0), position (0, 1), position (–1, 0), or position (0, –1). Picking a potential next step, say (1, 0), we find ourselves in the same type of situation we did before. To successfully traverse the maze from the new current position, we must successfully traverse it from an adjacent position. At any point, some of the adjacent positions may be invalid, may be blocked, or may represent a possible successful path. We continue this process recursively. If the base case, position (7, 12), is reached, the maze has been traversed successfully.

LISTING 7.2

```
/**
 * Maze represents a maze of characters. The goal is to get from the
 * top left corner to the bottom right, following a path of 1's.
 *
 * @author Dr. Chase
 * @author Dr. Lewis
 * @version 1.0, 8/18/08
 */

public class Maze
{
   private final int TRIED = 3;
   private final int PATH = 7;

   private int[][] grid = { {1,1,1,0,1,1,0,0,0,1,1,1,1},
                            {1,0,1,1,1,0,1,1,1,1,0,0,1},
                            {0,0,0,0,1,0,1,0,1,0,1,0,0},
                            {1,1,1,0,1,1,1,0,1,0,1,1,1},
                            {1,0,1,0,0,0,0,1,1,1,0,0,1},
                            {1,0,1,1,1,1,1,1,0,1,1,1,1},
                            {1,0,0,0,0,0,0,0,0,0,0,0,0},
                            {1,1,1,1,1,1,1,1,1,1,1,1,1}};

   /**
    * Attempts to recursively traverse the maze. Inserts special
    * characters indicating locations that have been tried and that
    * eventually become part of the solution.
    *
    * @param row     the integer value of the row
    * @param column  the integer value of the column
    * @return        true if the maze has been solved
    */
   public boolean traverse (int row, int column)
   {
      boolean done = false;

      if (valid (row, column))
      {
         grid[row][column] = TRIED; // this cell has been tried

         if (row == grid.length-1 && column == grid[0].length-1)
            done = true; // the maze is solved
         else
         {
```

LISTING 7.2 *continued*

```
            done = traverse (row+1, column);     // down
            if (!done)
               done = traverse (row, column+1);  // right
            if (!done)
               done = traverse (row-1, column);  // up
            if (!done)
               done = traverse (row, column-1);  // left
         }

         if (done) // this location is part of the final path
            grid[row][column] = PATH;

      }

      return done;
   }
   /**
    * Determines if a specific location is valid.
    *
    * @param row      the column to be checked
    * @param column   the column to be checked
    * @return         true if the location is valid
    */
   private boolean valid (int row, int column)
   {
      boolean result = false;

      // check if cell is in the bounds of the matrix
      if (row >= 0 && row < grid.length &&
         column >= 0 && column < grid[row].length)

         // check if cell is not blocked and not previously tried
         if (grid[row][column] == 1)
            result = true;
      return result;
   }

   /**
    * Returns the maze as a string.
    *
    * @return a string representation of the maze
    */
   public String toString ()
   {
      String result = "\n";
```

LISTING 7.2 *continued*

```
      for (int row=0; row < grid.length; row++)
      {
         for (int column=0; column < grid[row].length; column++)
            result += grid[row][column] + "";

         result += "\n";
      }

      return result;
   }
}
```

The recursive method in the `Maze` class is called `traverse`. It returns a `boolean` value that indicates whether a solution was found. First the method determines if a move to the specified row and column is valid. A move is considered valid if it stays within the grid boundaries and if the grid contains a 1 in that location, indicating that a move in that direction is not blocked. The initial call to `traverse` passes in the upper-left location (0, 0).

If the move is valid, the grid entry is changed from a 1 to a 3, marking this location as visited so that later we don't retrace our steps. Then the `traverse` method determines if the maze has been completed by having reached the bottom-right location. Therefore, there are actually three possibilities of the base case for this problem that will terminate any particular recursive path:

- An invalid move because the move is out of bounds or blocked
- An invalid move because the move has been tried before
- A move that arrives at the final location

If the current location is not the bottom-right corner, we search for a solution in each of the primary directions, if necessary. First, we look down by recursively calling the `traverse` method and passing in the new location. The logic of the `traverse` method starts all over again using this new position. A solution either is ultimately found by first attempting to move down from the current location, or it is not found. If it's not found, we try moving right. If that fails, we try moving up. Finally, if no other direction has yielded a correct path, we try moving left. If no direction from the current location yields a correct solution, then there is no path from this location, and `traverse` returns false. If the very first invocation of the `traverse` method returns false, then there is no possible path through this maze.

If a solution is found from the current location, then the grid entry is changed to a 7. The first 7 is placed in the bottom-right corner. The next 7 is placed in the location that led to the bottom-right corner, and so on until the final 7 is placed in the upper-left corner. Therefore, when the final maze is printed, 0 still indicates a blocked path, 1 indicates an open path that was never tried, 3 indicates a path that was tried but failed to yield a correct solution, and 7 indicates a part of the final solution of the maze.

Note that there are several opportunities for recursion in each call to the `traverse` method. Any or all of them might be followed, depending on the maze configuration. Although there may be many paths through the maze, the recursion terminates when a path is found. Carefully trace the execution of this code while following the maze array to see how the recursion solves the problem. Then consider the difficulty of producing a nonrecursive solution.

The Towers of Hanoi

The *Towers of Hanoi* puzzle was invented in the 1880s by Edouard Lucas, a French mathematician. It has become a favorite among computer scientists because its solution is an excellent demonstration of recursive elegance.

The puzzle consists of three upright pegs (towers) and a set of disks with holes in the middle so that they slide onto the pegs. Each disk has a different diameter. Initially, all of the disks are stacked on one peg in order of size such that the largest disk is on the bottom, as shown in Figure 7.6.

The goal of the puzzle is to move all of the disks from their original (first) peg to the destination (third) peg. We can use the "extra" peg as a temporary place to put disks, but we must obey the following three rules:

- We can move only one disk at a time.
- We cannot place a larger disk on top of a smaller disk.
- All disks must be on some peg except for the disk in transit between pegs.

These rules imply that we must move smaller disks "out of the way" in order to move a larger disk from one peg to another. Figure 7.7 shows the step-by-step

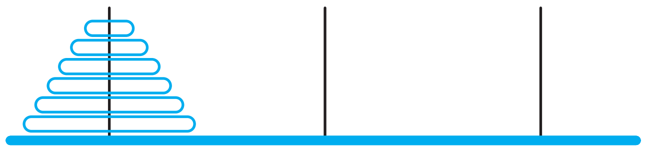


FIGURE 7.6 The Towers of Hanoi puzzle

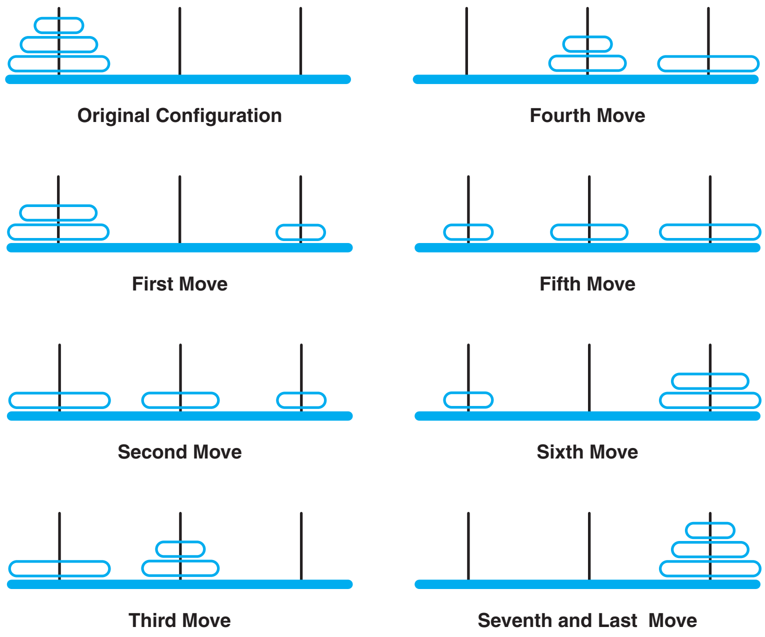


FIGURE 7.7 A solution to the three-disk Towers of Hanoi puzzle

solution for the Towers of Hanoi puzzle using three disks. To move all three disks from the first peg to the third peg, we first have to get to the point where the smaller two disks are out of the way on the second peg so that the largest disk can be moved from the first peg to the third peg.

The first three moves shown in Figure 7.7 can be thought of as "moving the smaller disks out of the way." The fourth move puts the largest disk in its final place. The last three moves put the smaller disks in their final place on top of the largest one.

Let's use this idea to form a general strategy. To move a stack of N disks from the original peg to the destination peg:

- Move the topmost N–1 disks from the original peg to the extra peg.
- Move the largest disk from the original peg to the destination peg.
- Move the N–1 disks from the extra peg to the destination peg.

This strategy lends itself nicely to a recursive solution. The step to move the N–1 disks out of the way is the same problem all over again: moving a stack of disks. For this subtask, though, there is one less disk, and our destination peg is what we were originally calling the extra peg. An analogous situation occurs after

LISTING 7.3

```
/**
 * SolveTowers demonstrates recursion.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 8/18/08
 */
public class SolveTowers
{

   /**
    * Creates a TowersOfHanoi puzzle and solves it.
    */
   public static void main (String[] args)
   {
      TowersOfHanoi towers = new TowersOfHanoi (4);
      towers.solve();
   }
}
```

we have moved the largest disk, and we have to move the original N–1 disks again.

The base case for this problem occurs when we want to move a "stack" that consists of only one disk. That step can be accomplished directly and without recursion.

The program in Listing 7.3 creates a `TowersOfHanoi` object and invokes its `solve` method. The output is a step-by-step list of instructions that describes how the disks should be moved to solve the puzzle. This example uses four disks, which is specified by a parameter to the `TowersOfHanoi` constructor.

The `TowersOfHanoi` class, shown in Listing 7.4, uses the `solve` method to make an initial call to `moveTower`, the recursive method. The initial call indicates that all of the disks should be moved from peg 1 to peg 3, using peg 2 as the extra position.

The `moveTower` method first considers the base case (a "stack" of one disk). When that occurs, it calls the `moveOneDisk` method, which prints a single line describing that particular move. If the stack contains more than one disk, we call `moveTower` again to get the N–1 disks out of the way, then move the largest disk, then move the N–1 disks to their final destination with yet another call to `moveTower`.

LISTING 7.4

```
/**
 * TowersOfHanoi represents the classic Towers of Hanoi puzzle.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 8/18/08
 */

public class TowersOfHanoi
{
   private int totalDisks;

   /**
    * Sets up the puzzle with the specified number of disks.
    *
    * @param disks the number of disks to start the towers puzzle with
    */
   public TowersOfHanoi (int disks)
   {
      totalDisks = disks;
   }

   /**
    * Performs the initial call to moveTower to solve the puzzle.
    * Moves the disks from tower 1 to tower 3 using tower 2.
    */
   public void solve()
   {
      moveTower (totalDisks, 1, 3, 2);
   }

   /**
    * Moves the specified number of disks from one tower to another
    * by moving a subtower of n-1 disks out of the way, moving one
    * disk, then moving the subtower back. Base case of 1 disk.
    *
    * @param numDisks  the number of disks to move
    * @param start     the starting tower
    * @param end       the ending tower
    * @param temp      the temporary tower
    */
```

LISTING 7.4 *continued*

```
   private void moveTower (int numDisks, int start, int end, int temp)
   {
      if (numDisks == 1)
         moveOneDisk (start, end);
      else
      {
         moveTower (numDisks-1, start, temp, end);
         moveOneDisk (start, end);
         moveTower (numDisks-1, temp, end, start);
      }
   }

   /**
    * Prints instructions to move one disk from the specified start
    * tower to the specified end tower.
    *
    * @param start  the starting tower
    * @param end    the ending tower
    */
   private void moveOneDisk (int start, int end)
   {
      System.out.println ("Move one disk from " + start + " to " +
                          end);
   }
}
```

Note that the parameters to `moveTower` describing the pegs are switched around as needed to move the partial stacks. This code follows our general strategy and uses the `moveTower` method to move all partial stacks. Trace the code carefully for a stack of three disks to understand the processing. Figure 7.8 shows the UML diagram for this problem.

7.4 Analyzing Recursive Algorithms

In Chapter 2, we explored the concept of analyzing an algorithm to determine its complexity (usually its time complexity) and expressed it in terms of a growth function. The growth function gave us the order of the algorithm, which can be used to compare it to other algorithms that accomplish the same task.

KEY CONCEPT
The order of a recursive algorithm can be determined using techniques similar to those used in analyzing iterative processing.

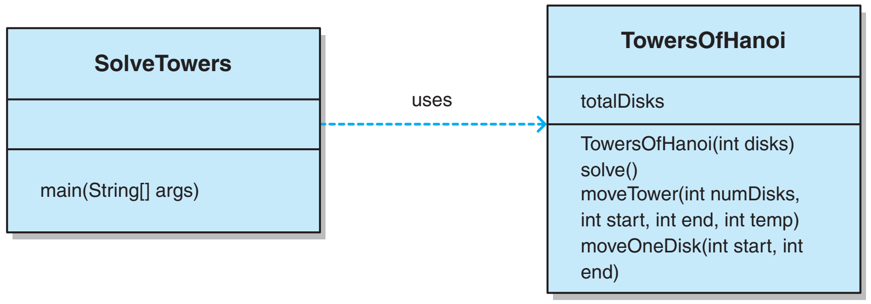


FIGURE 7.8 UML description of the `SolveTowers` and `TowersOfHanoi` classes

When analyzing a loop, we determined the order of the body of the loop and multiplied it by the number of times the loop was executed. Analyzing a recursive algorithm uses similar thinking. Determining the order of a recursive algorithm is a matter of determining the order of the recursion (the number of times the recursive definition is followed) and multiplying that by the order of the body of the recursive method.

Consider the recursive method presented in Section 7.2 that computes the sum of the integers from 1 to some positive value. We reprint it here for convenience:

```
// This method returns the sum of 1 to num
public int sum (int num)
{
   int result;
   if (num == 1)
      result = 1;
   else
      result = num + sum (num-1);
   return result;
}
```

The size of this problem is naturally expressed as the number of values to be summed. Because we are summing the integers from 1 to `num`, the number of values to be summed is `num`. The operation of interest is the act of adding two values together. The body of the recursive method performs one addition operation, and therefore is O(1). Each time the recursive method is invoked, the value of `num` is decreased by 1. Therefore, the recursive method is called `num` times, so the order of the recursion is O(n). Thus, because the body is O(1) and the recursion is O(n), the order of the entire algorithm is O(n).

We will see that in some algorithms the recursive step operates on half as much data as the previous call, thus creating an order of recursion of O(log n). If the body of the method is O(1), then the whole algorithm is O(log n). If the body of the method is O(n), then the whole algorithm is O(n log n).

Now consider the Towers of Hanoi puzzle. The size of the puzzle is naturally the number of disks, and the processing operation of interest is the step of moving one disk from one peg to another. Each call to the recursive method `moveTower` results in one disk being moved. Unfortunately, except for the base case, each recursive call results in calling itself *twice more*, and each call operates on a stack of disks that is only one less than the stack that is passed in as the parameter. Thus, calling `moveTower` with 1 disk results in 1 disk being moved, calling `moveTower` with 2 disks results in 3 disks being moved, calling `moveTower` with 3 disks results in 7 disks being moved, calling `moveTower` with 4 disks results in 15 disks being moved, etc. Looking at it another way, if `f(n)` is the growth function for this problem, then:

$$f(n) = 1 \text{ when n is equal to } 1$$

for $n > 1$,

$$\begin{aligned} f(n) &= 2*(f(n-1) + 1) \\ &= 2^n - 1 \end{aligned}$$

Contrary to its short and elegant implementation, the solution to the Towers of Hanoi puzzle is terribly inefficient. To solve the puzzle with a stack of n disks, we have to make $2^n - 1$ individual disk moves. Therefore, the Towers of Hanoi algorithm is $O(2^n)$. As we discussed in Chapter 2, this order is an example of exponential complexity. As the number of disks increases, the number of required moves increases exponentially.

KEY CONCEPT

The Towers of Hanoi solution has exponential complexity, which is very inefficient. Yet the implementation of the solution is remarkably short and elegant.

Legend has it that priests of Brahma are working on this puzzle in a temple at the center of the world. They are using 64 gold disks, moving them between pegs of pure diamond. The downside is that when the priests finish the puzzle, the world will end. The upside is that even if they move one disk every second of every day, it will take them over 584 billion years to complete it. That's with a puzzle of only 64 disks! It is certainly an indication of just how intractable exponential algorithm complexity is.

Summary of Key Concepts

- Recursion is a programming technique in which a method calls itself. A key to being able to program recursively is to be able to think recursively.
- Any recursive definition must have a nonrecursive part, called the base case, which permits the recursion to eventually end.
- Mathematical problems and formulas are often expressed recursively.
- Each recursive call to a method creates new local variables and parameters.
- A careful trace of recursive processing can provide insight into the way it is used to solve a problem.
- Recursion is the most elegant and appropriate way to solve some problems, but for others it is less intuitive than an iterative solution.
- The order of a recursive algorithm can be determined using techniques similar to those used in analyzing iterative processing.
- The Towers of Hanoi solution has exponential complexity, which is very inefficient. Yet the implementation of the solution is incredibly short and elegant.

Self-Review Questions

SR 7.1 What is recursion?

SR 7.2 What is infinite recursion?

SR 7.3 When is a base case needed for recursive processing?

SR 7.4 Is recursion necessary?

SR 7.5 When should recursion be avoided?

SR 7.6 What is indirect recursion?

SR 7.7 Explain the general approach to solving the Towers of Hanoi puzzle. How does it relate to recursion?

Exercises

EX 7.1 Write a recursive definition of a valid Java identifier.

EX 7.2 Write a recursive definition of x^y (x raised to the power y), where x and y are integers and $y > 0$.

EX 7.3 Write a recursive definition of i * j (integer multiplication), where $i > 0$. Define the multiplication process in terms of integer addition. For example, 4 * 7 is equal to 7 added to itself 4 times.

EX 7.4 Write a recursive definition of the Fibonacci numbers, a sequence of integers, each of which is the sum of the previous two numbers. The first two numbers in the sequence are 0 and 1. Explain why you would not normally use recursion to solve this problem.

EX 7.5 Modify the method that calculates the sum of the integers between 1 and N shown in this chapter. Have the new version match the following recursive definition: The sum of 1 to N is the sum of 1 to (N/2) plus the sum of (N/2 + 1) to N. Trace your solution using an N of 7.

EX 7.6 Write a recursive method that returns the value of N! (N factorial) using the definition given in this chapter. Explain why you would not normally use recursion to solve this problem.

EX 7.7 Write a recursive method to reverse a string. Explain why you would not normally use recursion to solve this problem.

EX 7.8 Design or generate a new maze for the `MazeSearch` program in this chapter, and rerun the program. Explain the processing in terms of your new maze, giving examples of a path that was tried but failed, a path that was never tried, and the ultimate solution.

EX 7.9 Annotate the lines of output of the `SolveTowers` program in this chapter to show the recursive steps.

EX 7.10 Produce a chart showing the number of moves required to solve the Towers of Hanoi puzzle using the following number of disks: 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, and 25.

EX 7.11 Determine and explain the order of your solution to Exercise 7.4.

EX 7.12 Determine and explain the order of your solution to Exercise 7.5.

EX 7.13 Determine and explain the order of your solution to Exercise 7.6.

EX 7.14 Determine the order of the recursive maze solution presented in this chapter.

Programming Projects

PP 7.1 Design and implement a program that implements Euclid's algorithm for finding the greatest common divisor of two positive integers. The greatest common divisor is the largest integer that divides both values without producing a remainder. In a class called `DivisorCalc`, define a static method called `gcd` that accepts two integers, `num1` and `num2`. Create a driver to test your implementation. The recursive algorithm is defined as follows:

```
gcd (num1, num2) is num2 if num2 <= num1 and num2 divides num1
gcd (num1, num2) is gcd (num2, num1) if num1 < num2
gcd (num1, num2) is gcd (num2, num1%num2) otherwise
```

PP 7.2 Modify the `Maze` class so that it prints out the path of the final solution as it is discovered, without storing it.

PP 7.3 Design and implement a program that traverses a 3D maze.

PP 7.4 Design and implement a recursive program that solves the Nonattacking Queens problem. That is, write a program to determine how eight queens can be positioned on an eight-by-eight chessboard so that none of them is in the same row, column, or diagonal as any other queen. There are no other chess pieces on the board.

PP 7.5 In the language of an alien race, all words take the form of Blurbs. A Blurb is a Whoozit followed by one or more Whatzits. A Whoozit is the character '`x`' followed by zero or more '`y`'s. A Whatzit is a '`q`' followed by either a '`z`' or a '`d`', followed by a Whoozit. Design and implement a recursive program that generates random Blurbs in this alien language.

PP 7.6 Design and implement a recursive program to determine if a string is a valid Blurb as defined in the previous project description.

PP 7.7 Design and implement a recursive program to determine and print the Nth line of Pascal's Triangle, as shown below. Each interior value is the sum of the two values above it. (*Hint*: Use an array to store the values on each line.)

```
                        1

                    1       1

                1       2       1

            1       3       3       1

        1       4       6       4       1

    1       5      10      10       5       1

1       6      15      20      15       6       1

    1       7      21      35      35      21       7       1

1       8      28      56      70      56      28       8       1
```

PP 7.8 Design and implement a graphic version of the Towers of Hanoi puzzle. Allow the user to set the number of disks used in the puzzle. The user should be able to interact with the puzzle in two main ways. The user can move the disks from one peg to another using the mouse, in which case the program should ensure that each move is legal. The user can also watch a solution take place as an animation, with pause/resume buttons. Permit the user to control the speed of the animation.

Answers to Self-Review Questions

SRA 7.1 Recursion is a programming technique in which a method calls itself, solving a smaller version of the problem each time, until the terminating condition is reached.

SRA 7.2 Infinite recursion occurs when there is no base case that serves as a terminating condition, or when the base case is improperly specified. The recursive path is followed forever. In a recursive program, infinite recursion will often result in an error that indicates that available memory has been exhausted.

SRA 7.3 A base case is always required to terminate recursion and begin the process of returning through the calling hierarchy. Without the base case, infinite recursion results.

SRA 7.4 Recursion is not necessary. Every recursive algorithm can be written in an iterative manner. However, some problem solutions are much more elegant and straightforward when written recursively.

SRA 7.5 Avoid recursion when the iterative solution is simpler and more easily understood and programmed. Recursion has the overhead of multiple method calls and is not always intuitive.

SRA 7.6 Indirect recursion occurs when a method calls another method, which calls another method, and so on until one of the called methods invokes the original. Indirect recursion is usually more difficult to trace than direct recursion, in which a method calls itself.

SRA 7.7 The Towers of Hanoi puzzle of N disks is solved by moving N–1 disks out of the way onto an extra peg, moving the largest disk to its destination, then moving the N–1 disks from the extra peg to the destination. This solution is inherently recursive because, to move the substack of N–1 disks, we can use the same process.

This page intentionally left blank

8 Sorting and Searching

Two common tasks in the world of software development are searching for a particular element within a group and sorting a group of elements into a particular order. There are a variety of algorithms that can be used to accomplish these tasks, and the differences between them are worth exploring carefully. These topics go hand in hand with our study of collections and data structures.

CHAPTER OBJECTIVES

- Examine the linear search and binary search algorithms
- Examine several sort algorithms
- Discuss the complexity of these algorithms

8.1 Searching

KEY CONCEPT
Searching is the process of finding a designated target within a group of items or determining that it doesn't exist.

KEY CONCEPT
An efficient search minimizes the number of comparisons made.

Searching is the process of finding a designated *target element* within a group of items, or determining that the target does not exist within the group. The group of items to be searched is sometimes called the *search pool*.

This section examines two common approaches to searching: a linear search and a binary search. Later in this book, other search techniques are presented that use the characteristics of particular data structures to facilitate the search process.

Our goal is to perform the search as efficiently as possible. In terms of algorithm analysis, we want to minimize the number of comparisons we have to make to find the target. In general, the more items there are in the search pool, the more comparisons it will take to find the target. Thus, the size of the problem is defined by the number of items in the search pool.

To be able to search for an object, we must be able to compare one object to another. Our implementations of these algorithms search an array of `Comparable` objects. Therefore, the elements involved must implement the `Comparable` interface and be comparable to each other. We might attempt to accomplish this restriction in the header for the `SortingandSearching` class in which all of our sorting and searching methods are located by doing something like:

```
public class SortingandSearching<T extends Comparable<? super T>>
```

The net effect of this generic declaration is that we can instantiate the `SortingandSearching` class with any class that implements the `Comparable` interface. Recall that the `Comparable` interface contains one method, `compareTo`, which is designed to return an integer that is less than zero, equal to zero, or greater than zero (respectively) if the object is less than, equal to, or greater than the object to which it is being compared. Therefore, any class that implements the `Comparable` interface defines the relative order of any two objects of that class.

Declaring the `SortingandSearching` in this manner, however, will cause us to have to instantiate the class any time we want to use one of the search or sort methods. This is awkward at best for a class that contains nothing but service methods. A better solution would be to declare all of the methods as static and generic. Let's first remind ourselves about the concept of static methods, and then we will explore generic static methods.

Static Methods

A *static method* (also called a *class method*) can be invoked through the class name (all the methods of the Math class are static methods, for example). You don't have to instantiate an object of the class to invoke a static method. For example, the sqrt method is called through the Math class as follows:

```
System.out.println ("Square root of 27: " + Math.sqrt(27));
```

A method is made static by using the static modifier in the method declaration. As we have seen, the main method of a Java program must be declared with the static modifier; this is so that main can be executed by the interpreter without instantiating an object from the class that contains main.

KEY CONCEPT

A method is made static by using the static modifier in the method declaration.

Because static methods do not operate in the context of a particular object, they cannot reference instance variables, which exist only in an instance of a class. The compiler will issue an error if a static method attempts to use a nonstatic variable. A static method can, however, reference static variables, because static variables exist independent of specific objects. Therefore, the main method can access only static or local variables.

The methods in the Math class perform basic computations based on values passed as parameters. There is no object state to maintain in these situations; therefore, there is no good reason to force us to create an object in order to request these services.

Generic Methods

Similar to what we have done in creating generic classes, we can also create generic methods. To create a generic method, we simply insert a generic declaration in the header of the method immediately preceding the return type.

```
public static <T extends Comparable<? super T>> boolean
   linearSearch (T[] data, int min, int max, T target)
```

It makes sense that the generic declaration has to come before the return type so that the generic type could be used as part of the return type.

Now that we can create a generic static method, we do not need to instantiate the SortingandSearching class each time we need one of the methods. Instead, we can simply invoke the static method using the class name and including our type to replace the generic type. For example, an invocation of the linearSearch method to search an array of Strings might look like this:

```
SortingandSearching.linearSearch(targetarray, min, max, target);
```

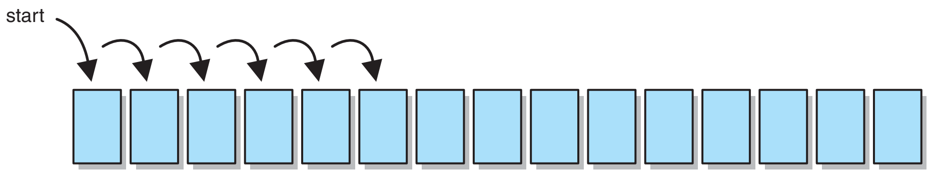


FIGURE 8.1 A linear search

Note that it is not necessary to specify the type to replace the generic type. The compiler will infer the type from the arguments provided. Thus for this line of code, the compiler will replace the generic type T with whatever the element type is for `targetarray` and the type of `target`.

Linear Search

If the search pool is organized into a list of some kind, one straightforward way to perform the search is to start at the beginning of the list and compare each value in turn to the target element. Eventually, we will either find the target or come to the end of the list and conclude that the target doesn't exist in the group. This approach is called a *linear search* because it begins at one end and scans the search pool in a linear manner. This process is depicted in Figure 8.1.

The following method implements a linear search. It accepts the array of elements to be searched, the beginning and ending index for the search, and the target value sought. The method returns a `boolean` value that indicates whether or not the target element was found.

```
/**
 * Searches the specified array of objects using a linear search
 * algorithm.
 *
 * @param data    the array to be sorted
 * @param min     the integer representation of the minimum value
 * @param max     the integer representation of the maximum value
 * @param target  the element being searched for
 * @return        true if the desired element is found
 */

public static <T extends Comparable<? super T>> boolean
   linearSearch (T[] data, int min, int max, T target)
```

```
{
   int index = min;
   boolean found = false;

   while (!found && index <= max)
   {
      if (data[index].compareTo(target) == 0)
         found = true;
      index++;
   }

   return found;
}
```

The `while` loop steps through the elements of the array, terminating when either the element is found or the end of the array is reached. The `boolean` variable `found` is initialized to `false` and is changed to `true` only if the target element is located.

Variations on this implementation could return the element found in the array if it is found and return a null reference if it is not found. Alternatively, an exception could be thrown if the target element is not found.

The `linearSearch` method could be incorporated into any class. Our version of this method is defined as part of a class containing methods that provide various searching capabilities.

The linear search algorithm is fairly easy to understand, though it is not particularly efficient. Note that a linear search does not require the elements in the search pool to be in any particular order within the array. The only criterion is that we must be able to examine them one at a time in turn. The binary search algorithm, described next, improves on the efficiency of the search process, but it only works if the search pool is ordered.

Binary Search

If the group of items in the search pool is sorted, then our approach to searching can be much more efficient than that of a linear search. A *binary search* algorithm eliminates large parts of the search pool with each comparison by capitalizing on the fact that the search pool is in sorted order.

KEY CONCEPT
A binary search capitalizes on the fact that the search pool is sorted.

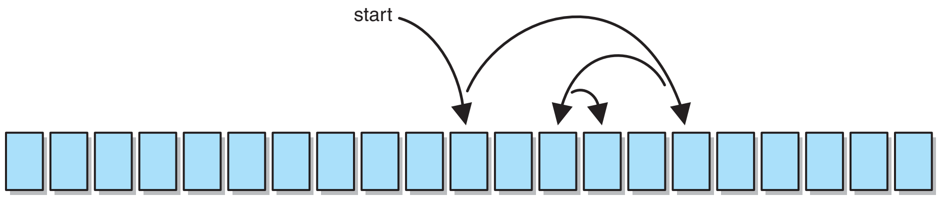


FIGURE 8.2 A binary search

Instead of starting the search at one end or the other, a binary search begins in the middle of the sorted list. If the target element is not found at that middle element, then the search continues. And because the list is sorted, we know that if the target is in the list, it will be on one side of the array or the other, depending on whether the target is less than or greater than the middle element. Thus, because the list is sorted, we eliminate half of the search pool with one carefully chosen comparison. The remaining half of the search pool represents the *viable candidates* in which the target element may yet be found.

The search continues in this same manner, examining the middle element of the viable candidates, eliminating half of them. Each comparison reduces the viable candidates by half until eventually the target element is found or there are no more viable candidates, which means the target element is not in the search pool. The process of a binary search is depicted in Figure 8.2.

Let's look at an example. Consider the following sorted list of integers:

10 12 18 22 31 34 40 46 59 67 69 72 80 84 98

Suppose we were trying to determine if the number 67 is in the list. Initially, the target could be anywhere in the list (all items in the search pool are viable candidates).

The binary search approach begins by examining the middle element, in this case 46. That element is not our target, so we must continue searching. But since we know that the list is sorted, we know that if 67 is in the list, it must be in the second half of the data, because all data items to the left of the middle have values of 46 or less. This leaves the following viable candidates to search (shown in bold):

10 12 18 22 31 34 40 46 **59 67 69 72 80 84 98**

Continuing the same approach, we examine the middle value of the viable candidates (72). Again, this is not our target value, so we must continue the search. This time we can eliminate all values higher than 72, leaving:

10 12 18 22 31 34 40 46 **59 67 69** 72 80 84 98

Note that, in only two comparisons, we have reduced the viable candidates from 15 items down to 3 items. Employing the same approach again, we select the middle element, 67, and find the element we are seeking. If it had not been our target, we would have continued with this process until we either found the value or eliminated all possible data.

KEY CONCEPT

A binary search eliminates half of the viable candidates with each comparison.

With each comparison, a binary search eliminates approximately half of the remaining data to be searched (it also eliminates the middle element as well). That is, a binary search eliminates half of the data with the first comparison, another quarter of the data with the second comparison, another eighth of the data with the third comparison, and so on.

The following method implements a binary search. Like the `linearSearch` method, it accepts an array of `Comparable` objects to be searched as well as the target value. It also takes integer values representing the minimum index and maximum index that define the portion of the array to search (the viable candidates).

```java
/**
 * Searches the specified array of objects using a binary search
 * algorithm.
 *
 * @param data    the array to be sorted
 * @param min     the integer representation of the minimum value
 * @param max     the integer representation of the maximum value
 * @param target  the element being searched for
 * @return        true if the desired element is found
 */

public static <T extends Comparable<? super T>> boolean
   binarySearch (T[] data, int min, int max, T target)
{
   boolean found = false;
   int midpoint = (min + max) / 2;  // determine the midpoint

   if (data[midpoint].compareTo(target) == 0)
      found = true;

   else if (data[midpoint].compareTo(target) > 0)
   {
      if (min <= midpoint - 1)
         found = binarySearch(data, min, midpoint - 1, target);
   }
   else if (midpoint + 1 <= max)
      found = binarySearch(data, midpoint + 1, max, target);

   return found;
}
```

Note that the `binarySearch` method is implemented recursively. If the target element is not found, and there is more data to search, the method calls itself, passing parameters that shrink the size of viable candidates within the array. The `min` and `max` indexes are used to determine if there is still more data to search. That is, if the reduced search area does not contain at least one element, the method does not call itself, and a value of false is returned.

At any point in this process, we may have an even number of values to search, and therefore two "middle" values. As far as the algorithm is concerned, the midpoint used could be either of the two middle values as long as the same choice is made consistently. In this implementation of the binary search, the calculation that determines the midpoint index discards any fractional part, and therefore picks the first of the two middle values.

Comparing Search Algorithms

For a linear search, the best case occurs when the target element happens to be the first item we examine in the group. The worst case occurs when the target is not in the group, and we have to examine every element before we determine that it isn't present. The expected case is that we would have to search half of the list before we find the element. That is, if there are n elements in the search pool, on average we would have to examine n/2 elements before finding the one for which we were searching.

Therefore, the linear search algorithm has a linear time complexity of O(n). Because the elements are searched one at a time in turn, the complexity is linear—in direct proportion to the number of elements to be searched.

A binary search, on the other hand, is generally much faster. Because we can eliminate half of the remaining data with each comparison, we can find the element much more quickly. The best case is that we find the target in one comparison—that is, the target element happens to be at the midpoint of the array. The worst case occurs if the element is not present in the list, in which case we have to make approximately $\log_2 n$ comparisons before we eliminate all of the data. Thus, the expected case for finding an element that is in the search pool is approximately $(\log_2 n)/2$ comparisons.

Therefore, a binary search is a *logarithmic algorithm* and has a time complexity of $O(\log_2 n)$. Compared to a linear search, a binary search is much faster for large values of n.

KEY CONCEPT

A binary search has logarithmic complexity, making it very efficient for a large search pool.

The question might be asked, if a logarithmic search is more efficient than a linear search, why would we ever use a linear search? First, a linear search is generally simpler than a binary search, and it is therefore easier to program and debug. Second, a linear search does not require the additional overhead of sorting the search list. There is a trade-off between the effort to keep the search pool sorted and the efficiency of the search.

For small problems, there is little practical difference between the two types of algorithms. However, as n gets larger, the binary search becomes increasingly attractive. Suppose a given set of data contains one million elements. In a linear search, we would have to examine each of the one million elements to determine that a particular target element is not in the group. In a binary search, we could make that conclusion in roughly 20 comparisons.

8.2 Sorting

Sorting is the process of arranging a group of items into a defined order, either ascending or descending, based on some criteria. For example, you may want to alphabetize a list of names or put a list of survey results into descending numeric order.

KEY CONCEPT

Sorting is the process of arranging a list of items into a defined order based on some criteria.

Many sort algorithms have been developed and critiqued over the years. In fact, sorting is considered to be a classic area of study in computer science. Similar to search algorithms, sort algorithms generally are divided into two categories based on efficiency: *sequential sorts*, which typically use a pair of nested loops and require roughly n^2 comparisons to sort n elements, and *logarithmic sorts*, which typically require roughly $n\log_2 n$ comparisons to sort n elements. As with the search algorithms, when n is small, there is little practical difference between the two categories of algorithms.

In this chapter, we examine three sequential sorts—selection sort, insertion sort, and bubble sort—and two logarithmic sorts—quick sort and merge sort. Other search techniques are examined elsewhere in the book based on particular data structures.

Before we dive into particular sort algorithms, let's look at a general sorting problem to solve. The `SortPhoneList` program, shown in Listing 8.1, creates an array of `Contact` objects, sorts those objects, and then prints the sorted list. In this implementation, the `Contact` objects are sorted using a call to the `selectionSort` method, which we examine later in this chapter. However, any sorting method described in this chapter could be used instead to achieve the same results.

LISTING 8.1

```
/**
 * SortPhoneList driver for testing an object selection sort.
 *
 * @author Dr. Chase
 * @author Dr. Lewis
 * @version 1.0, 8/18/08
 */

public class SortPhoneList
{
   /**
    * Creates an array of Contact objects, sorts them, then prints
    * them.
    */

   public static void main (String[] args)
   {
      Contact[] friends = new Contact[7];

      friends[0] = new Contact ("John", "Smith", "610-555-7384");
      friends[1] = new Contact ("Sarah", "Barnes", "215-555-3827");
      friends[2] = new Contact ("Mark", "Riley", "733-555-2969");
      friends[3] = new Contact ("Laura", "Getz", "663-555-3984");
      friends[4] = new Contact ("Larry", "Smith", "464-555-3489");
      friends[5] = new Contact ("Frank", "Phelps", "322-555-2284");
      friends[6] = new Contact ("Marsha", "Grant", "243-555-2837");

      SortingAndSearching.selectionSort(friends);

      for (int index = 0; index < friends.length; index++)
         System.out.println (friends[index]);
   }
}
```

Each `Contact` object represents a person with a last name, a first name, and a phone number. The `Contact` class is shown in Listing 8.2. The UML description of these classes is left as an exercise.

The `Contact` class implements the `Comparable` interface and therefore provides a definition of the `compareTo` method. In this case, the contacts are

LISTING 8.2

```
/**
 * Contact represents a phone contact.
 *
 * @author Dr. Chase
 * @author Dr. Lewis
 * @version 1.0, 8/18/08
 */

public class Contact implements Comparable
{
   private String firstName, lastName, phone;

   /**
    * Sets up this contact with the specified information.
    *
    * @param first      a string representation of a first name
    * @param last       a string representation of a last name
    * @param telephone  a string representation of a phone number
    */

   public Contact (String first, String last, String telephone)
   {
      firstName = first;
      lastName = last;
      phone = telephone;
   }

   /**
    * Returns a description of this contact as a string.
    *
    * @return  a string representation of this contact
    */

   public String toString ()
   {
      return lastName + ", " + firstName + "\t" + phone;
   }

   /**
    * Uses both last and first names to determine lexical ordering.
    *
    * @param other  the contact to be compared to this contact
    * @return       the integer result of the comparison
    */
```

LISTING 8.2 *continued*

```
   public int compareTo (Object other)
   {
      int result;

      if (lastName.equals(((Contact)other).lastName))
         result = firstName.compareTo(((Contact)other).firstName);
      else
         result = lastName.compareTo(((Contact)other).lastName);

      return result;
   }
}
```

sorted by last name; if two contacts have the same last name, their first names are used.

Now let's examine several sort algorithms and their implementations. Any of these could be used to put the `Contact` objects into sorted order.

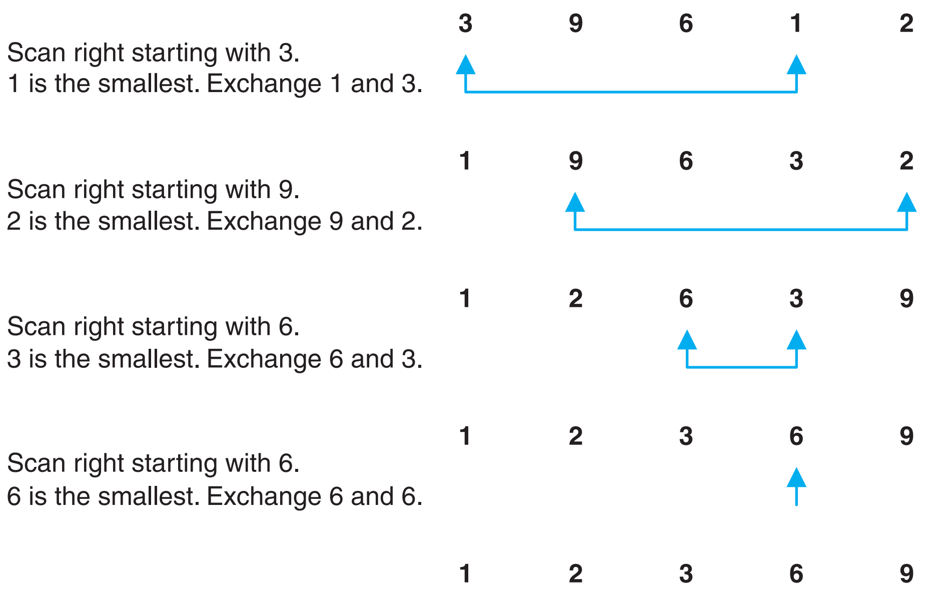
Selection Sort

The *selection sort* algorithm sorts a list of values by repetitively putting a particular value into its final, sorted, position. In other words, for each position in the list, the algorithm selects the value that should go in that position and puts it there.

KEY CONCEPT
The selection sort algorithm sorts a list of values by repetitively putting a particular value into its final, sorted, position.

The general strategy of the selection sort algorithm is as follows: Scan the entire list to find the smallest value. Exchange that value with the value in the first position of the list. Scan the rest of the list (all but the first value) to find the smallest value, and then exchange it with the value in the second position of the list. Scan the rest of the list (all but the first two values) to find the smallest value, and then exchange it with the value in the third position of the list. Continue this process for each position in the list. When complete, the list is sorted. The selection sort process is illustrated in Figure 8.3.

The following method defines an implementation of the selection sort algorithm. It accepts an array of objects as a parameter. When it returns to the calling method, the elements within the array are sorted.



Scan right starting with 3.
1 is the smallest. Exchange 1 and 3.
Scan right starting with 9.
2 is the smallest. Exchange 9 and 2.
Scan right starting with 6.
3 is the smallest. Exchange 6 and 3.
Scan right starting with 6.
6 is the smallest. Exchange 6 and 6.
3 9 6 1 2
1 9 6 3 2
1 2 6 3 9
1 2 3 6 9
1 2 3 6 9

FIGURE 8.3 Illustration of selection sort processing

```
/**
 * Sorts the specified array of integers using the selection
 * sort algorithm.
 *
 * @param data the array to be sorted
 */

public static <T extends Comparable<? super T>> void selectionSort (T[] data)
{
   int min;
   T temp;

   for (int index = 0; index < data.length-1; index++)
   {
      min = index;
      for (int scan = index+1; scan < data.length; scan++)
         if (data[scan].compareTo(data[min])<0)
            min = scan;

      // Swap the values
```

```
         temp = data[min];
         data[min] = data[index];
         data[index] = temp;
      }
   }
```

The implementation of the `selectionSort` method uses two loops to sort an array. The outer loop controls the position in the array where the next smallest value will be stored. The inner loop finds the smallest value in the rest of the list by scanning all positions greater than or equal to the index specified by the outer loop. When the smallest value is determined, it is exchanged with the value stored at `index`. This exchange is accomplished by three assignment statements using an extra variable called `temp`. This type of exchange is called *swapping*.

Note that because this algorithm finds the smallest value during each iteration, the result is an array sorted in ascending order (i.e., smallest to largest). The algorithm can easily be changed to put values in descending order by finding the largest value each time.

Insertion Sort

The *insertion sort* algorithm sorts a list of values by repetitively inserting a particular value into a subset of the list that has already been sorted. One at a time, each unsorted element is inserted at the appropriate position in that sorted subset until the entire list is in order.

KEY CONCEPT

The insertion sort algorithm sorts a list of values by repetitively inserting a particular value into a subset of the list that has already been sorted.

The general strategy of the insertion sort algorithm is as follows: Sort the first two values in the list relative to each other by exchanging them if necessary. Insert the list's third value into the appropriate position relative to the first two (sorted) values. Then insert the fourth value into its proper position relative to the first three values in the list. Each time an insertion is made, the number of values in the sorted subset increases by one. Continue this process until all values in the list are completely sorted. The insertion process requires that the other values in the array shift to make room for the inserted element. Figure 8.4 illustrates the insertion sort process.

The following method implements an insertion sort:

```
/**
 * Sorts the specified array of objects using an insertion
 * sort algorithm.
 *
 * @param data the array to be sorted
 */

public static <T extends Comparable<? super T>> void insertionSort (T[] data)
{
   for (int index = 1; index < data.length; index++)
   {
      T key = data[index];
      int position = index;

      // Shift larger values to the right

      while (position > 0 && data[position-1].compareTo(key) > 0)
      {
         data[position] = data[position-1];
         position-;
      }
      data[position] = key;
   }
}
```

Similar to the selection sort implementation, the `insertionSort` method uses two loops to sort an array of objects. In the insertion sort, however, the outer loop controls the index in the array of the next value to be inserted. The inner loop compares the current insert value with values stored at lower indexes (which make up a

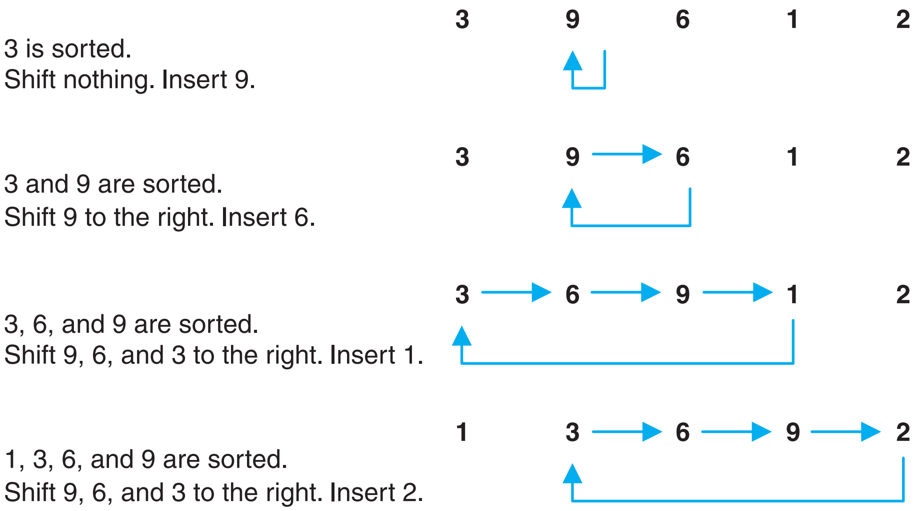

3 9 6 1 2
3 is sorted.
Shift nothing. Insert 9.
3 9 6 1 2
3 and 9 are sorted.
Shift 9 to the right. Insert 6.
3 6 9 1 2
3, 6, and 9 are sorted.
Shift 9, 6, and 3 to the right. Insert 1.
1 3 6 9 2
1, 3, 6, and 9 are sorted.
Shift 9, 6, and 3 to the right. Insert 2.

FIGURE 8.4 Illustration of insertion sort processing

sorted subset of the entire list). If the current insert value is less than the value at `position`, then that value is shifted to the right. Shifting continues until the proper position is opened to accept the insert value. Each iteration of the outer loop adds one more value to the sorted subset of the list, until the entire list is sorted.

Bubble Sort

A *bubble sort* is another sequential sort algorithm that uses two nested loops. It sorts values by repeatedly comparing neighboring elements in the list and swapping their position if they are not in order relative to each other.

KEY CONCEPT

The bubble sort algorithm sorts a list by repeatedly comparing neighboring elements and swapping them if necessary.

The general strategy of the bubble sort algorithm is as follows: Scan through the list comparing adjacent elements and exchange them if they are not in relative order. This has the effect of "bubbling" the largest value to the last position in the list, which is its appropriate position in the final sorted list. Then scan through the list again, bubbling up the second-to-last value. This process continues until all elements have been bubbled into their correct positions.

Each pass through the bubble sort algorithm moves the largest value to its final position. A pass may also reposition other elements as well. For example, if we started with the list:

9 6 8 12 3 1 7

we would first compare 9 and 6 and, finding them not in the correct order, swap them, yielding:

6 9 8 12 3 1 7

Then we would compare 9 to 8 and, again, finding them not in the correct order, swap them, yielding:

6 8 9 12 3 1 7

Then we would compare 9 to 12. Since they are in the correct order, we don't swap them. Instead, we move to the next pair of values to compare. That is, we then compare 12 to 3. Because they are not in order, we swap them, yielding:

6 8 9 3 12 1 7

We then compare 12 to 1 and swap them, yielding:

6 8 9 3 1 12 7

We then compare 12 to 7 and swap them, yielding:

6 8 9 3 1 7 12

This completes one pass through the data to be sorted. After this first pass, the largest value in the list (12) is in its correct position, but we cannot be sure about any of the other numbers. Each subsequent pass through the data guarantees that one more element is put into the correct position. Thus we make n–1 passes through the data, because if n–1 elements are in the correct, sorted positions, the nth item must also be in the correct location.

An implementation of the bubble sort algorithm is shown in the following method:

```
/**
 * Sorts the specified array of objects using a bubble sort
 * algorithm.
 *
 * @param data the array to be sorted
 */

public static <T extends Comparable<? super T>> void bubbleSort (T[] data)
{
   int position, scan;
   T temp;

   for (position = data.length - 1; position >= 0; position--)
   {
      for (scan = 0; scan <= position - 1; scan++)
      {
         if (data[scan].compareTo(data[scan+1]) > 0)
         {

            // Swap the values

            temp = data[scan];
            data[scan] = data[scan + 1];
            data[scan + 1] = temp;
         }
      }
   }
}
```

The outer `for` loop in the `bubbleSort` method represents the n–1 passes through the data. The inner `for` loop scans through the data, performs the pairwise comparisons of the neighboring data, and swaps them if necessary.

Note that the outer loop also has the effect of decreasing the position that represents the maximum index to examine in the inner loop. That is, after the first pass, which puts the last value in its correct position, there is no need to consider that value in future passes through the data. After the second pass, we can forget about the last two, and so on. Thus the inner loop examines one less value on each pass.

Quick Sort

The sort algorithms we have discussed thus far in this chapter (selection sort, insertion sort, and bubble sort) are relatively simple, but they are inefficient sequential sorts that use a pair of nested loops and require roughly n^2 comparisons to sort a list of n elements. Now we can turn our attention to more efficient sorts that lend themselves to a recursive implementation.

KEY CONCEPT

The quick sort algorithm sorts a list by partitioning the list and then recursively sorting the two partitions.

The *quick sort* algorithm sorts a list by partitioning the list using an arbitrarily chosen *partition element* and then recursively sorting the sublists on either side of the partition element. The general strategy of the quick sort algorithm is as follows: First, choose one element of the list to act as a partition element. Next, partition the list so that all elements less than the partition element are to the left of that element and all elements greater than the partition element are to the right. Finally, apply this quick sort strategy (recursively) to both partitions.

If the data is truly random, the choice of the partition element is arbitrary. We will use the element in the middle of the section we want to partition. For efficiency reasons, it would be nice if the partition element divided the list roughly in half, but the algorithm will work no matter what element is chosen as the partition.

Let's look at an example of creating a partition. If we started with the following list:

305 65 7 90 120 110 8

we would choose 90 as our partition element. We would then rearrange the list, swapping the elements that are less than 90 to the left side and those that are greater than 90 to the right side, yielding:

8 65 7 90 120 110 305

We would then apply the quick sort algorithm separately to both partitions. This process continues until a partition contains only one element, which is inherently sorted. Thus, after the algorithm is applied recursively to either side, the entire list is sorted. Once the initial partition element is determined and placed, it is never considered or moved again.

The following method implements the quick sort algorithm. It accepts an array of objects to sort and the minimum and maximum index values used for a particular call to the method. For the initial call to the method, the values of `min` and `max` would encompass the entire set of elements to be sorted.

```
/**
 * Sorts the specified array of objects using the quick sort
 * algorithm.
 *
 * @param data    the array to be sorted
 * @param min     the integer representation of the minimum value
 * @param max     the integer representation of the maximum value
 */

public static <T extends Comparable<? super T>> void
   quickSort (T[] data, int min, int max)
{
   int indexofpartition;

   if (max - min > 0)
   {

     // Create partitions

     indexofpartition = findPartition(data, min, max);

     // Sort the left side

     quickSort(data, min, indexofpartition - 1);

     // Sort the right side

     quickSort(data, indexofpartition + 1, max);
   }
}
```

The `quickSort` method relies heavily on the `findPartition` method, which it calls initially to divide the sort area into two partitions. The `findPartition` method returns the index of the partition value. Then the `quickSort` method is called twice (recursively) to sort the two partitions. The base case of the recursion, represented by the `if` statement in the `quickSort` method, is a list of one element or less, which is already inherently sorted. The `findPartition` method is shown below:

```
/**
 * Used by the quick sort algorithm to find the partition.
 *
 * @param data    the array to be sorted
 * @param min     the integer representation of the minimum value
 * @param max     the integer representation of the maximum value
 */

private static <T extends Comparable<? super T>> int
   findPartition (T[] data, int min, int max)
{
   int left, right;
   T temp, partitionelement;
   int middle = (min + max)/2;

   partitionelement = data[middle]; // use middle element as partition
   left = min;
   right = max;

   while (left<right)
   {
      // search for an element that is > the partitionelement

      while (data[left].compareTo(partitionelement) <=0 &&
                            left < right)
         left++;

      // search for an element that is < the partitionelement

      while (data[right].compareTo(partitionelement) > 0)
         right--;

      // swap the elements

      if (left<right)
      {
         temp = data[left];
         data[left] = data[right];
         data[right] = temp;
      }
  }

   // move partition element to partition index

   temp = data[min];
   data[min] = data[right];
   data[right] = temp;

   return right;
}
```

The two inner `while` loops of the `findPartition` method are used to find elements to swap that are in the wrong partitions. The first loop scans from left to right looking for an element that is greater than the partition element. The second loop scans from right to left looking for an element that is less than the partition element. When these two elements are found, they are swapped. This process continues until the right and left indexes meet in the "middle" of the list. The location where they meet also indicates where the partition element (which isn't moved from its initial location until the end) will ultimately reside.

What happens if we get a poor partition element? If the partition element is near the smallest or the largest element in the list, then we effectively waste a pass through the data. One way to ensure a better partition element is to choose the middle of three elements. For example, the algorithm could check the first, middle, and last elements in the list and choose the middle value as the partition element. This approach is left as a programming project.

Merge Sort

The *merge sort* algorithm, another recursive sort algorithm, sorts a list by recursively dividing the list in half until each sublist has one element and then recombining these sublists in order.

The general strategy of the merge sort algorithm is as follows: Begin by dividing the list in two roughly equal parts and then recursively calling itself with each of those lists. Continue the recursive decomposition of the list until the base case of the recursion is reached, where the list is divided into lists of length one, which are by definition sorted. Then, as control passes back up the recursive calling structure, the algorithm merges the two sorted sublists resulting from the two recursive calls into one sorted list.

KEY CONCEPT
The merge sort algorithm sorts a list by recursively dividing the list in half until each sublist has one element and then merging these sublists into the sorted order.

For example, if we started with the initial list from our example in the previous section, the recursive decomposition portion of the algorithm would yield the results shown in Figure 8.5.

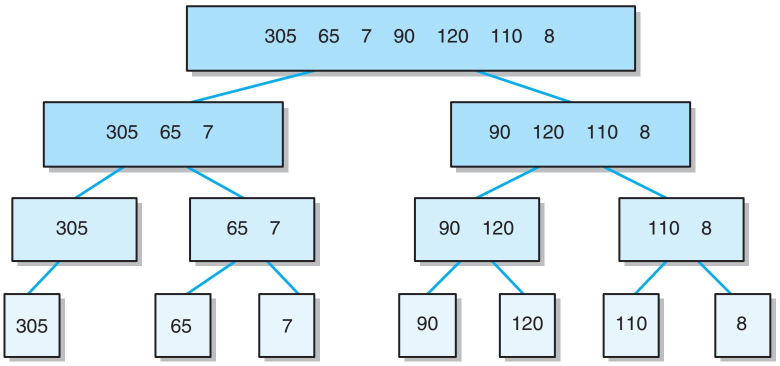


FIGURE 8.5 The decomposition of merge sort

The merge portion of the algorithm would then recombine the list as shown in Figure 8.6.

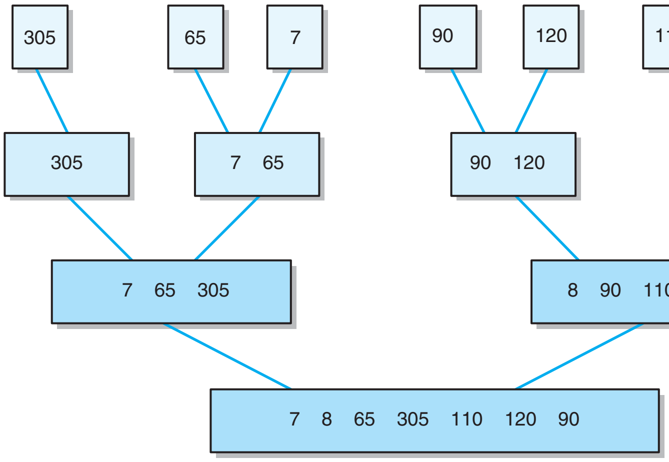


FIGURE 8.6 The merge portion of the merge sort algorithm

An implementation of the merge sort algorithm is shown below:

```
/**
 * Sorts the specified array of objects using the merge sort
 * algorithm.
 *
 * @param data    the array to be sorted
 * @param min     the integer representation of the minimum value
 * @param max     the integer representation of the maximum value
 */

public static <T extends Comparable<? super T>> void
   mergeSort (T[] data, int min, int max)
{
   T[] temp;
   int index1, left, right;

   // return on list of length one

   if (min==max)
      return;

   // find the length and the midpoint of the list

   int size = max - min + 1;
   int pivot = (min + max) / 2;
   temp = (T[])(new Comparable[size]);
```

```
      mergeSort(data, min, pivot);    // sort left half of list
      mergeSort(data, pivot + 1, max);    // sort right half of list

      // copy sorted data into workspace

      for (index1 = 0; index1 < size; index1++)
         temp[index1] = data[min + index1];

      // merge the two sorted lists

      left = 0;
      right = pivot - min + 1;
      for (index1 = 0; index1 < size; index1++)
      {
          if (right <= max - min)
             if (left <= pivot - min)
                if (temp[left].compareTo(temp[right]) > 0)
                   data[index1 + min] = temp[right++];

                else
                   data[index1 + min] = temp[left++];
             else
                data[index1 + min] = temp[right++];
          else
             data[index1 + min] = temp[left++];
      }
   }
```

8.3 Radix Sort

To this point, all of the sorting techniques we have discussed have involved comparing elements within the list to each other. As we have seen, the best of these comparison-based sorts is O(nlogn). What if there was a way to sort elements without comparing them directly to each other. It might then be possible to build a more efficient sorting algorithm. We can find such a technique by revisiting our discussion of queues from Chapter 5.

KEY CONCEPT
A radix sort is inherently based on queue processing.

A sort is based on some particular value, called the *sort key*. For example, a set of people might be sorted by their last name. A *radix sort*, rather than comparing items by sort key, is based on the structure of the sort key. Separate queues are created for each possible value of each digit or character of the sort key. The number of queues, or the number of possible values, is called the *radix*. For example, if we were sorting strings made up of lowercase alphabetic characters, the radix would be 26. We would use 26 separate queues, one for each possible character. If we were sorting decimal numbers, then the radix would be ten, one for each digit 0 through 9.

Let's look at an example that uses a radix sort to put ten three-digit numbers in order. To keep things manageable, we will restrict the digits of these numbers to 0 through 5, which means we will need only six queues.

Each three-digit number to be sorted has a 1s position (right digit), a 10s position (middle digit), and a 100s position (left digit). The radix sort will make three passes through the values, one for each digit position. On the first pass, each number is put on the queue corresponding to its 1s digit. On the second pass, each number is put on the queue corresponding to its 10s digit. And finally, on the third pass, each number is put on the queue corresponding to its 100s digit.

Originally, the numbers are loaded into the queues from the original list. On the second pass, the numbers are taken from the queues in a particular order. They are retrieved from the digit 0 queue first, and then the digit 1 queue, etc. For each queue, the numbers are processed in the order in which they come off the queue. This processing order is crucial to the operation of a radix sort. Likewise, on the third pass, the numbers are again taken from the queues in the same way. When the numbers are pulled off of the queues after the third pass, they will be completely sorted.

Figure 8.7 shows the processing of a radix sort for ten three-digit numbers. The number 442 is taken from the original list and put onto the queue corresponding to

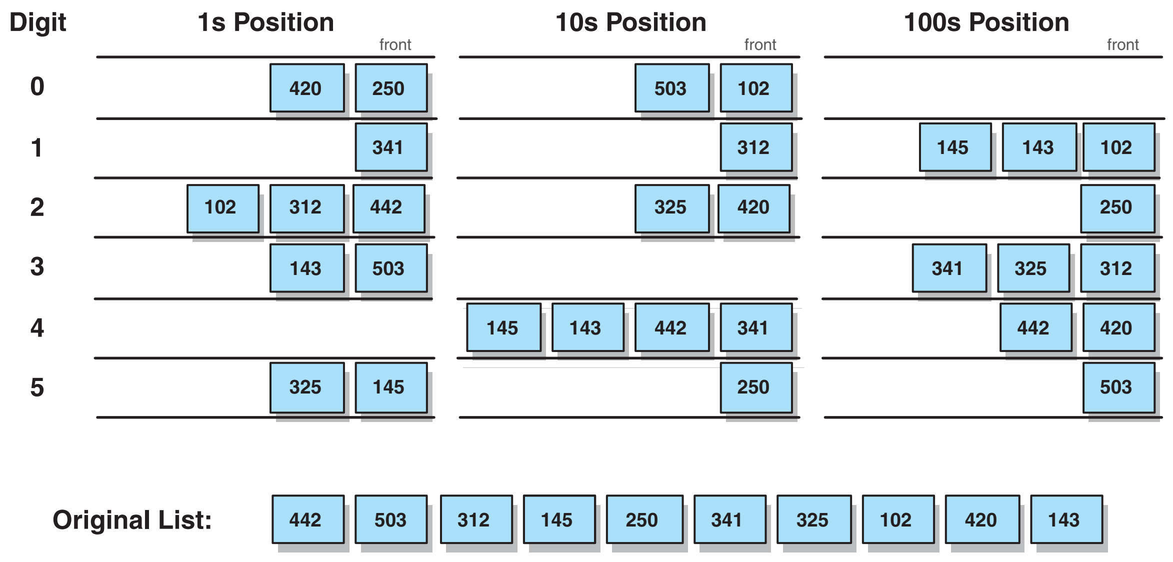


FIGURE 8.7 A radix sort of ten three-digit numbers

digit 2. Then 503 is put onto the queue corresponding to digit 3. Then 312 is put onto the queue corresponding to digit 2 (following 442). This continues for all values, resulting in the set of queues for the 1s position.

Assume, as we begin the second pass, that we have a fresh set of six empty digit queues. In actuality, the queues can be used over again if processed carefully. To begin the second pass, the numbers are taken from the 0 digit queue first. The number 250 is put onto the queue for digit 5, and then 420 is put onto the queue for digit 2. Then we can move to the next queue, taking 341 and putting it onto the queue for digit 4. This continues until all numbers have been taken off of the 1s position queues, resulting in the set of queues for the 10s position.

For the third pass, the process is repeated. First, 102 is put onto the queue for digit 1, then 503 is put onto the queue for digit 5, then 312 is put onto the queue for digit 3. This continues until we have the final set of digit queues for the 100s position. These numbers are now in sorted order if taken off of each queue in turn.

Let's now look at a program that implements the radix sort. For this example, we will sort four-digit numbers, and we won't restrict the digits used in those numbers. Listing 8.3 shows the `RadixSort` class, which contains a single `main`

LISTING 8.3

```java
/**
 * RadixSort driver demonstrates the use of queues in the execution of a radix
 * sort.
 *
 * @author Dr. Chase
 * @author Dr. Lewis
 * @version 1.0, 8/18/08
 */

 import jss2.ArrayQueue;

 public class RadixSort
 {
    /**
     *  Performs a radix sort on a set of numeric values.
     **/

    public static void main (String[] args)
    {
       int[] list = {7843, 4568, 8765, 6543, 7865, 4532, 9987, 3241,
                     6589, 6622, 1211};
```

LISTING 8.3 *continued*

```
         String temp;
         Integer numObj;
         int digit, num;

         ArrayQueue<Integer>[] digitQueues = (ArrayQueue<Integer>[])
           (new ArrayQueue[10]);

         for (int digitVal = 0; digitVal <= 9; digitVal++)
            digitQueues[digitVal] = new ArrayQueue<Integer>();

         // sort the list

         for (int position=0; position <= 3; position++)
         {
            for (int scan=0; scan < list.length; scan++)
            {
               temp = String.valueOf (list[scan]);
               digit = Character.digit (temp.charAt(3-position), 10);
               digitQueues[digit].enqueue (new Integer(list[scan]));
            }

            // gather numbers back into list

            num = 0;
            for (int digitVal = 0; digitVal <= 9; digitVal++)
            {
               while (!(digitQueues[digitVal].isEmpty()))
               {
                  numObj = digitQueues[digitVal].dequeue();
                  list[num] = numObj.intValue();
                  num++;
               }
            }
         }

         // output the sorted list

         for (int scan=0; scan < list.length; scan++)
            System.out.println (list[scan]);
    }
}
```

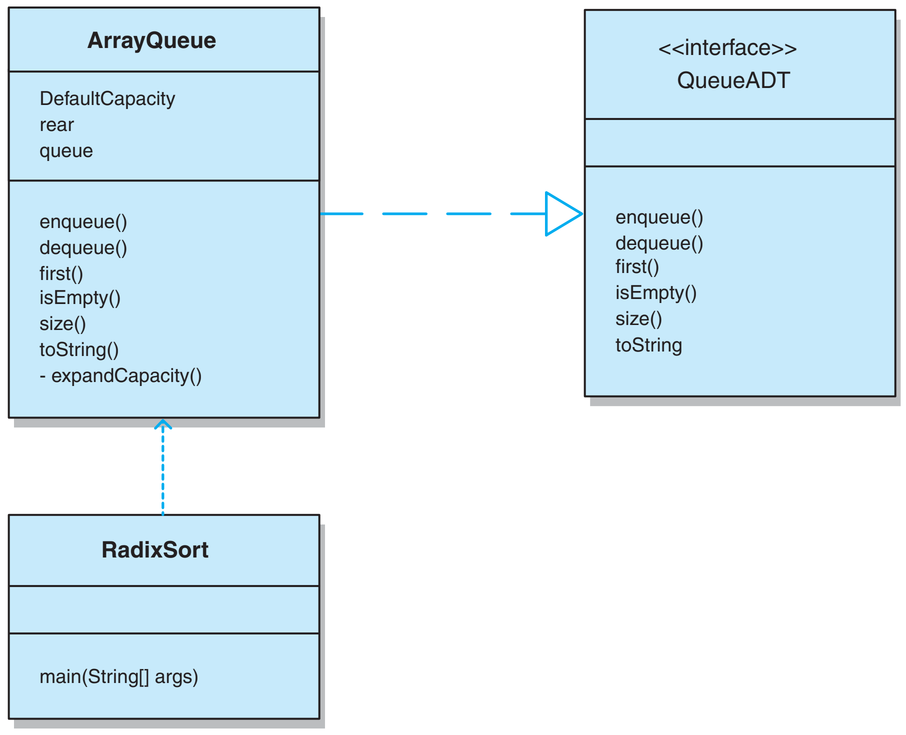


FIGURE 8.8 UML description of the `RadixSort` program

method. Using an array of ten queue objects (one for each digit 0 through 9), this method carries out the processing steps of a radix sort. Figure 8.8 shows the UML description of the `RadixSort` class.

Notice that in the `RadixSort` program, we cannot create an array of queues to hold integers because of the restriction that prevents the instantiation of arrays of generic types. Instead, we create an array of `CircularArrayQueues` and then cast that as an array of `CircularArrayQueue<Integer>`.

In the `RadixSort` program, the numbers are originally stored in an array called `list`. After each pass, the numbers are pulled off of the queues and stored back into the `list` array in the proper order. This allows the program to reuse the original array of ten queues for each pass of the sort.

The concept of a radix sort can be applied to any type of data as long as the sort key can be dissected into well-defined positions. Note that unlike the sorts we discussed earlier in this chapter, it's not reasonable to create a generic radix sort for any object, because dissecting the key values is an integral part of the process.

So what is the time complexity of a radix sort? In this case, there is not any comparison or swapping of elements. Elements are simply removed from a queue and placed in another one on each pass. For any given radix, the number of passes through the data is a constant based on the number of characters in the key; let's call it c. Then the time complexity of the algorithm is simply c*n. Keep in mind, from our discussion in Chapter 2, that we ignore constants when computing the time complexity of an algorithm. Thus the radix sort algorithm is O(n). So why not use radix sort for all of our sorting? First, each radix sort algorithm has to be designed specifically for the key of a given problem. Second, for keys where the number of digits in the key (c) and the number of elements in the list (n) are very close together, the actual time complexity of the radix sort algorithm mimics n^2 instead of n. In addition, we also need to keep in mind that there is another constant that impacts space complexity and that is the radix, or the number of possible values for each position or character in the key. Imagine, for example, trying to implement a radix sort for a key that allows any character from the Unicode character set. Because this set has more than 100,000 characters, we would need that many queues!

Summary of Key Concepts

- Searching is the process of finding a designated target within a group of items or determining that it doesn't exist.
- An efficient search minimizes the number of comparisons made.
- A method is made static by using the static modifier in the method declaration.
- A binary search capitalizes on the fact that the search pool is sorted.
- A binary search eliminates half of the viable candidates with each comparison.
- A binary search has logarithmic complexity, making it very efficient for a large search pool.
- Sorting is the process of arranging a list of items into a defined order based on some criteria.
- The selection sort algorithm sorts a list of values by repetitively putting a particular value into its final, sorted, position.
- The insertion sort algorithm sorts a list of values by repetitively inserting a particular value into a subset of the list that has already been sorted.
- The bubble sort algorithm sorts a list by repeatedly comparing neighboring elements and swapping them if necessary.
- The quick sort algorithm sorts a list by partitioning the list and then recursively sorting the two partitions.
- The merge sort algorithm sorts a list by recursively dividing the list in half until each sublist has one element and then merging these sublists into the sorted order.
- A radix sort is inherently based on queue processing.

Self-Review Questions

SR 8.1 When would a linear search be preferable to a logarithmic search?

SR 8.2 Which searching method requires that the list be sorted?

SR 8.3 When would a sequential sort be preferable to a recursive sort?

SR 8.4 The insertion sort algorithm sorts using what technique?

SR 8.5 The bubble sort algorithm sorts using what technique?

SR 8.6 The selection sort algorithm sorts using what technique?

SR 8.7 The quick sort algorithm sorts using what technique?

SR 8.8 The merge sort algorithm sorts using what technique?

SR 8.9 How many queues would it take to use a radix sort to sort names stored as all lowercase?

Exercises

EX 8.1 Compare and contrast the `linearSearch` and `binarySearch` algorithms by searching for the numbers 45 and 54 in the following list (3, 8, 12, 34, 54, 84, 91, 110).

EX 8.2 Using the list from Exercise 8.1, construct a table showing the number of comparisons required to sort that list for each of the sort algorithms (selection sort, insertion sort, bubble sort, quick sort, and merge sort).

EX 8.3 Using the same list from Exercise 8.1, what happens to the number of comparisons for each of the sort algorithms if the list is already sorted?

EX 8.4 Given the following list:

90 8 7 56 123 235 9 1 653

Show a trace of execution for:

a. selection sort

b. insertion sort

c. bubble sort

d. quick sort

e. merge sort

EX 8.5 Given the resulting sorted list from Exercise 8.4, show a trace of execution for a binary search, searching for the number 235.

EX 8.6 Draw the UML description of the `SortPhoneList` example.

EX 8.7 Hand trace a radix sort for the following list of five-digit student ID numbers, assuming that each digit must be between 1 and 5:

13224
32131
54355
12123
22331
21212
33333
54312

EX 8.8 What is the time complexity of a radix sort?

Programming Projects

PP 8.1 The bubble sort algorithm shown in this chapter is less efficient than it could be. If a pass is made through the list without exchanging any elements, this means that the list is sorted and there is no reason to continue. Modify this algorithm so that it will stop as soon as it recognizes that the list is sorted. *Do not* use a `break` statement!

PP 8.2 There is a variation of the bubble sort algorithm called a *gap sort* that, rather than comparing neighboring elements each time through the list, compares elements that are some number (i) positions apart, where i is an integer less than n. For example, the first element would be compared to the (i + 1) element, the second element would be compared to the (i + 2) element, the nth element would be compared to the (n − i) element, etc. A single iteration is completed when all of the elements that can be compared, have been compared. On the next iteration, i is reduced by some number greater than 1 and the process continues until i is less than 1. Implement a gap sort.

PP 8.3 Modify the sorts listed in the chapter (selection sort, insertion sort, bubble sort, quick sort, and merge sort) by adding code to each to tally the total number of comparisons and total execution time of each algorithm. Execute the sort algorithms against the same list, recording information for the total number of comparisons and total execution time for each algorithm. Try several different lists, including at least one that is already in sorted order.

PP 8.4 Modify the quick sort method to choose the partition element using the middle of three technique described in the chapter. Run this new version against the old version for several sets of data and compare the total execution time.

Answers to Self-Review Questions

SRA 8.1 A linear search would be preferable for relatively small, unsorted lists, and in languages where recursion is not supported.

SRA 8.2 Binary search.

SRA 8.3 A sequential sort would be preferable for relatively small data sets and in languages where recursion is not supported.

SRA 8.4 The insertion sort algorithm sorts a list of values by repetitively inserting a particular value into a subset of the list that has already been sorted.

SRA 8.5 The bubble sort algorithm sorts a list by repeatedly comparing neighboring elements in the list and swapping their position if they are not already in order.

SRA 8.6 The selection sort algorithm, which is an $O(n^2)$ sort algorithm, sorts a list of values by repetitively putting a particular value into its final, sorted, position.

SRA 8.7 The quick sort algorithm sorts a list by partitioning the list using an arbitrarily chosen partition element and then recursively sorting the sublists on either side of the partition element.

SRA 8.8 The merge sort algorithm sorts a list by recursively dividing the list in half until each sublist has one element and then recombining these sublists in order.

SRA 8.9 It would require 27 queues, one for each of the 26 letters in the alphabet and 1 to store the whole list before, during, and after sorting.

9 Trees

This chapter begins our exploration of nonlinear collections and data structures. We discuss the use and implementation of trees, define the terms associated with trees, analyze possible tree implementations, and look at examples of implementing and using trees.

CHAPTER OBJECTIVES

- Define trees as data structures
- Define the terms associated with trees
- Discuss the possible implementations of trees
- Analyze tree implementations of collections
- Discuss methods for traversing trees
- Examine a binary tree example

9.1 Trees

The collections we have examined to this point in the book (stacks, queues, and lists) are all linear data structures, meaning their elements are arranged in order one after another. A *tree* is a nonlinear structure in which elements are organized into a hierarchy. This section describes trees in general and establishes some important terminology.

KEY CONCEPT
A tree is a nonlinear structure whose elements are organized into a hierarchy.

Conceptually, a tree is composed of a set of *nodes* in which elements are stored and *edges* that connect one node to another. Each node is at a particular *level* in the tree hierarchy. The *root* of the tree is the only node at the top level of the tree. There is only one root node in a tree. Figure 9.1 shows a tree that helps to illustrate these terms.

The nodes at lower levels of the tree are the *children* of nodes at the previous level. In Figure 9.1, the nodes labeled B, C, D, and E are the children of A. Nodes F and G are the children of B. A node can have only one parent, but a node may have multiple children. Nodes that have the same parent are called *siblings*. Thus, nodes H, I, and J are siblings because they are all children of D.

KEY CONCEPT
Trees are described by a large set of related terms.

The root node is the only node in a tree that does not have a parent. A node that does not have any children is called a *leaf*. A node that is not the root and has at least one child is called an *internal node*. Note that the tree analogy is upside-down. Our trees "grow" from the root at the top of the tree to the leaves toward the bottom of the tree.

The root is the entry point into a tree structure. We can follow a *path* through the tree from parent to child. For example, the path from node A to N in Figure 9.1 is A, D, I, N. A node is the *ancestor* of another node if it is above it on the path

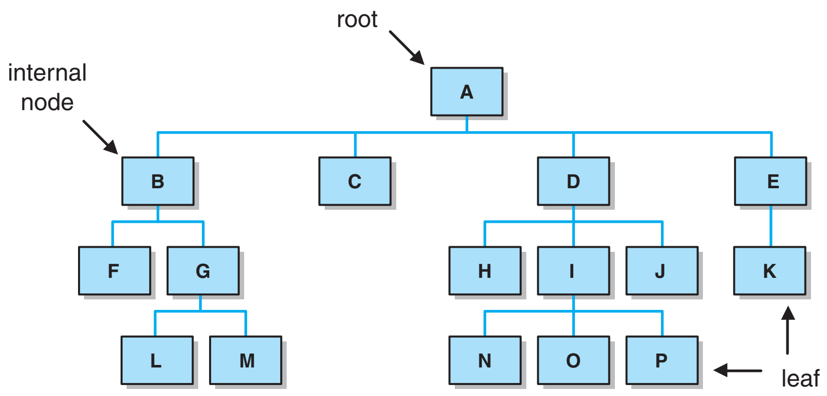


FIGURE 9.1 Tree terminology

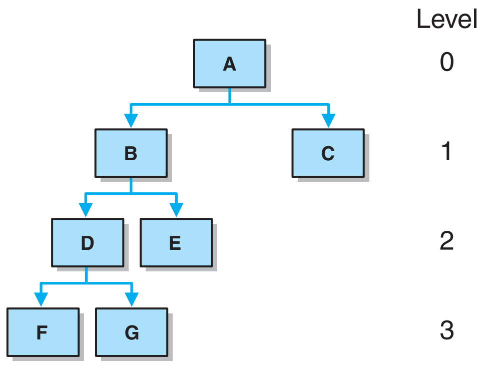


FIGURE 9.2 Path length and level

from the root. Thus the root is the ultimate ancestor of all nodes in a tree. Nodes that can be reached by following a path from a particular node are the *descendants* of that node.

The level of a node is also the length of the path from the root to the node. This *path length* is determined by counting the number of edges that must be followed to get from the root to the node. The root is considered to be level 0, the children of the root are at level 1, the grandchildren of the root are at level 2, and so on. Path length and level are depicted in Figure 9.2.

The *height* of a tree is the length of the longest path from the root to a leaf. Thus the height or order of the tree in Figure 9.2 is 3, because the path length from the root to leaves F and G is 3. The path length from the root to leaf C is 1.

Tree Classifications

Trees can be classified in many ways. The most important criterion is the maximum number of children any node in the tree may have. This value is sometimes referred to as the *order* of the tree. A tree that has no limit to the number of children a node may have is called a *general tree*. A tree that limits each node to no more than n children is referred to as an *n-ary tree*.

One n-ary tree is of particular importance. A tree in which nodes may have at most two children is called a *binary tree*. This type of tree is helpful in many situations. Much of our exploration of trees will focus on binary trees.

Another way to classify a tree is whether it is balanced or not. There are many definitions of balance depending upon the algorithms being used. We will explore some of these algorithms in the next chapter. Roughly speaking, a tree is considered to be *balanced* if all of the leaves of the tree are on the same level or at least within one level of each other. Thus, the tree shown on the left in Figure 9.3 is

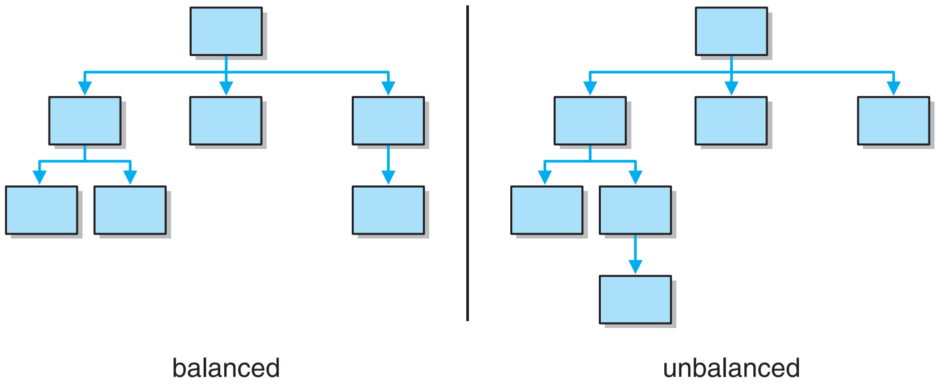


FIGURE 9.3 Balanced and unbalanced trees

balanced, while the one on the right is not. A balanced n-ary tree with m elements will have a height of $\log_n m$. Thus a balanced binary tree with n nodes will have a height of $\log_2 n$.

The concept of a complete tree is related to the balance of a tree. A tree is considered *complete* if it is balanced and all of the leaves at the bottom level are on the left side of the tree. While this is a seemingly arbitrary concept, this definition has implications for how the tree is stored in certain implementations. Another way to define this concept is that a complete binary tree has 2^k nodes at every level k except the last, where the nodes must be leftmost.

Another related concept is the notion of a full tree. An n-ary tree is considered *full* if all the leaves of the tree are at the same level and every node is either a leaf or has exactly n children. The balanced tree from Figure 9.3 would not be considered complete while of the 3-ary (or tertiary) trees shown in Figure 9.4, the trees shown in parts (a) and (c)–are complete. Only the third tree (c) in Figure 9.4 is full.

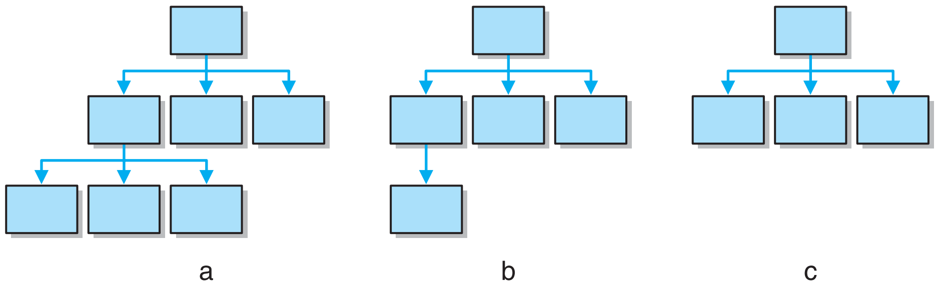


FIGURE 9.4 Some complete trees

9.2 Strategies for Implementing Trees

Let's examine some general strategies for implementing trees. The most obvious implementation of a tree is a linked structure. Each node could be defined as a `TreeNode` class, similar to what we did with the `LinearNode` class for linked lists. Each node would contain a pointer to the element to be stored in that node as well as pointers for each of the possible children of the node. Depending on the implementation, it may also be useful for each node to store a pointer to its parent. This use of pointers is similar to the concept of a doubly linked list where each node points not only to the next node in the list but to the previous one as well.

Because a tree is a nonlinear structure, it may not seem reasonable to attempt to implement it using an underlying linear structure such as an array. However, sometimes that approach is useful. The strategies for array implementations of a tree may be less obvious. There are two principle approaches: a computational strategy and a simulated link strategy.

Computational Strategy for Array Implementation of Trees

For certain types of trees, specifically binary trees, a computational strategy can be used for storing a tree using an array. One possible strategy is as follows: For any element stored in position n of the array, that element's left child will be stored in position (2 * n + 1) and that element's right child will be stored in position (2 * (n + 1)). This strategy is very effective and can be managed in terms of capacity in much the same way that we managed capacity for the array implementations of lists, queues, and stacks. However, despite the conceptual elegance of this solution, it is not without drawbacks. For example, if the tree that we are storing is not complete or is only relatively complete, we may be wasting large amounts of memory allocated in the array for positions of the tree that do not contain data. The computational strategy is illustrated in Figure 9.5.

KEY CONCEPT

One possible computational strategy places the left child of element n at position (2 * n + 1) and the right child at position (2 * (n + 1)).

Simulated Link Strategy for Array Implementation of Trees

A second possible array implementation of trees is modeled after the way operating systems manage memory. Instead of assigning elements of the tree to array positions by location in the tree, array positions are allocated contiguously on a first-come, first-served basis. Each element of the array will be a node class similar to the `TreeNode` class that we discussed earlier. However, instead of storing object reference variables as pointers to its children (and perhaps its parent), each node

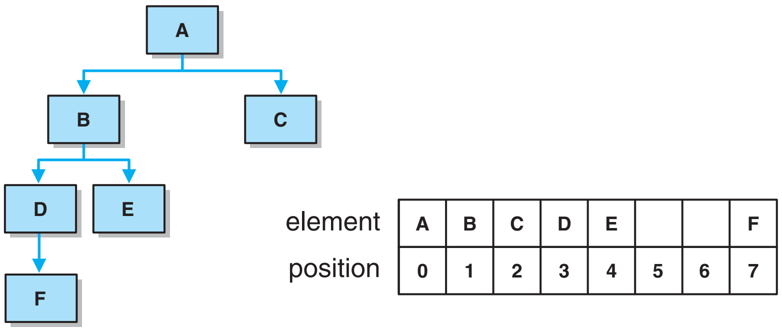


FIGURE 9.5 Computational strategy for array implementation of trees

would store the array index of each child (and perhaps its parent). This approach allows elements to be stored contiguously in the array so that space is not wasted. However, this approach increases the overhead for deleting elements in the tree, because it requires either that remaining elements be shifted to maintain contiguity or that a *freelist* be maintained. This strategy is illustrated in Figure 9.6. The order of the elements in the array is determined simply by their entry order into the tree. In this case, the entry order is assumed to have been A, C, B, E, D, F.

KEY CONCEPT

The simulated link strategy allows array positions to be allocated contiguously regardless of the completeness of the tree.

This same strategy may also be used when tree structures need to be stored directly on disk using a direct I/O approach. In this case, rather than using an array index as a pointer, each node will store the relative position in the file of its children so that an offset can be calculated given the base address of the file.

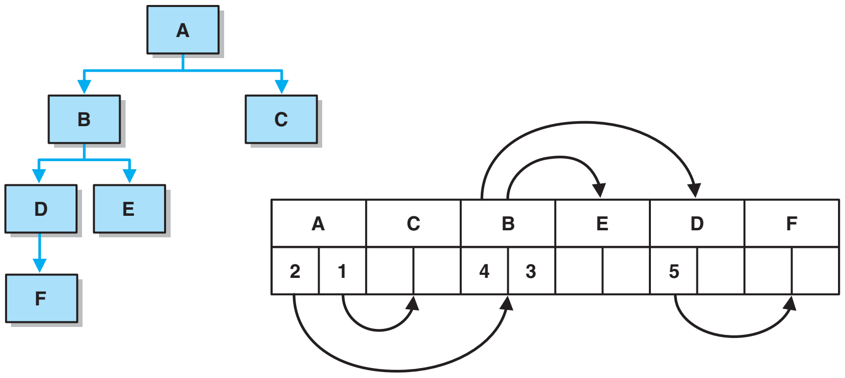


FIGURE 9.6 Simulated link strategy for array implementation of trees

DESIGN FOCUS

When does it begin to make sense to define an ADT for a collection? At this point, we have defined many of the terms for a tree and we have a general understanding of how a tree might be used, but are we ready to define an ADT? Not really. Trees, in the general sense, are more of an abstract data structure than a collection, so attempting to define an ADT for a general tree likely won't be very useful. Instead, we will wait until we have specified more details about the type of the tree and its use before we attempt to define an interface.

Analysis of Trees

As we discussed earlier, trees are a useful and efficient way to implement other collections. Let's consider an ordered list as an example. In our analysis of list implementations in Chapter 6, we described the `find` operation as expected case n/2 or O(n). However, if we were to implement an ordered list using a balanced *binary search tree*—a binary tree with the added property that the left child is always less than the parent, which is always less than or equal to the right child—then we could improve the efficiency of the `find` operation to O(log n). We will discuss binary search trees in much greater detail in Chapter 10.

KEY CONCEPT

In general, a balanced n-ary tree with m elements will have height $\log_n m$.

This increased efficiency is due to the fact that the height of such a tree will always be $\log_2 n$, where n is the number of elements in the tree. This is very similar to our discussion of the binary search in Chapter 8. In fact, for any balanced n-ary tree with m elements, the tree's height will be $\log_n m$. With the added ordering property of a binary search tree, you are guaranteed to, at worst, search one path from the root to a leaf and that path can be no longer than $\log_n m$.

DESIGN FOCUS

If trees provide more efficient implementations than linear structures, why would we ever use linear structures? There is an overhead associated with trees in terms of maintaining the structure and order of the tree that may not be present in other structures; thus there is a trade-off between this overhead and the size of the problem. With a relatively small n, the difference between the analysis of tree implementations and that of linear structures is not particularly significant relative to the overhead involved in the tree. However, as n increases, the efficiency of a tree becomes more attractive.

9.3 Tree Traversals

Because a tree is a nonlinear structure, the concept of traversing a tree is generally more interesting than the concept of traversing a linear structure. There are four basic methods for traversing a tree:

- *Preorder traversal*, which is accomplished by visiting each node, followed by its children, starting with the root
- *Inorder traversal*, which is accomplished by visiting the left child of the node, then the node, then any remaining nodes, starting with the root
- *Postorder traversal*, which is accomplished by visiting the children, then the node, starting with the root
- *Level-order traversal*, which is accomplished by visiting all of the nodes at each level, one level at a time, starting with the root

KEY CONCEPT
There are four basic methods for traversing a tree: preorder, inorder, postorder, and level-order.

Each of these definitions applies to all trees. However, as an example, let us examine how each of these definitions would apply to a binary tree (i.e., a tree in which each node has at most two children).

Preorder Traversal

Given the tree shown in Figure 9.7, a preorder traversal would produce the sequence A, B, D, E, C. The definition stated previously says that preorder traversal is accomplished by visiting each node, followed by its children, starting with the root. So, starting with the root, we visit the root, giving us A. Next we traverse to the first child of the root, which is the node containing B. We then use the same algorithm by first visiting the current node, yielding B, and then visiting its children. Next we traverse to the first child of B, which is the node containing D. We then use the same algorithm again by first visiting the current node, yielding D, and then visiting its children. Only this time, there are no children. We then traverse to any other children of B. This yields E, and because E has no children,

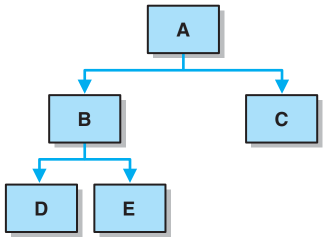


FIGURE 9.7 A complete tree

we then traverse to any other children of A. This brings us to the node containing C, where we again use the same algorithm, first visiting the node, yielding C, and then visiting any children. Because there are no children of C and no more children of A, the traversal is complete.

Stated in pseudocode for a binary tree, the algorithm for a preorder traversal is

```
Visit node
Traverse(left child)
Traverse(right child)
```

KEY CONCEPT

Preorder traversal means visit the node, then the left child, then the right child.

Inorder Traversal

Given the tree shown in Figure 9.7, an inorder traversal would produce the sequence D, B, E, A, C. As defined earlier, inorder traversal is accomplished by visiting the left child of the node, then the node, then any remaining nodes, starting with the root. So, starting with the root, we traverse to the left child of the root, the node containing B. We then use the same algorithm again and traverse to the left child of B, the node containing D. Note that we have not yet visited any nodes. Using the same algorithm again, we attempt to traverse to the left child of D. Because there is not one, we then visit the current node, yielding D. Continuing the same algorithm, we attempt to traverse to any remaining children of D. Because there are no children, we then visit the previous node, yielding B. We then attempt to traverse to any remaining children of B. This brings us to the node containing E. Because E does not have a left child, we visit the node, yielding E. Because E has no right child, we then visit the previous node, yielding A. We then traverse to any remaining children of A, which takes us to the node containing C. Using the same algorithm, we then attempt to traverse to the left child of C. Because there is not one, we then visit the current node, yielding C. We then attempt to traverse to any remaining children of C. Because there are no children, we return to the previous node, which happens to be the root. Because there are no more children of the root, the traversal is complete.

Stated in pseudocode for a binary tree, the algorithm for an inorder traversal is

```
Traverse(left child)
Visit node
Traverse(right child)
```

KEY CONCEPT

Inorder traversal means visit the left child, then the node, then the right child.

Postorder Traversal

Given the tree shown in Figure 9.7, a postorder traversal would produce the sequence D, E, B, C, A. As previously defined, postorder traversal is accomplished by visiting the children, then the node, starting with the root. So, starting from the root, we traverse to the left child, the node containing B. Repeating that process, we traverse to the left child again, the node containing D. Because that node does not have any children, we then visit that node, yielding D. Returning to the previous node, we visit the right child, the node containing E. Because this node does not have any children, we visit the node, yielding E, and then return to the previous node and visit it, yielding B. Returning to the previous node, in this case the root, we find that it has a right child, so we traverse to the right child, the node containing C. Because this node does not have any children, we visit it, yielding C. Returning to the previous node (the root), we find that it has no remaining children, so we visit it, yielding A, and the traversal is complete.

KEY CONCEPT

Postorder traversal means visit the left child, then the right child, then the node.

Stated in pseudocode for a binary tree, the algorithm for a postorder traversal is

```
Traverse(left child)
Traverse(right child)
Visit node
```

Level-Order Traversal

Given the tree shown in Figure 9.7, a level-order traversal would produce the sequence A, B, C, D, E. As defined earlier, a level-order traversal is accomplished by visiting all of the nodes at each level, one level at a time, starting with the root. Using this definition, we first visit the root, yielding A. Next we visit the left child of the root, yielding B, then the right child of the root, yielding C, and then the children of B, yielding D and E.

Stated in pseudocode for a binary tree, an algorithm for a level-order traversal is

```
Create a queue called nodes
Create an unordered list called results
Enqueue the root onto the nodes queue
While the nodes queue is not empty
{
  Dequeue the first element from the queue
  If that element is not null
    Add that element to the rear of the results list
    Enqueue the children of the element on the nodes queue
  Else
    Add null on the result list
}
Return an iterator for the result list
```

This algorithm for a level-order traversal is only one of many possible solutions. However, it does have some interesting properties. First, note that we are using collections, namely a queue and a list, to solve a problem within another collection, namely a binary tree. Second, recall that in our earlier discussions of iterators, we talked about their behavior with respect to the collection if the collection is modified while the iterator is in use. In this case, using a list to store the elements in the proper order and then returning an iterator over the list, this iterator behaves like a snap-shot of the binary tree and is not affected by any concurrent modifications. This can be both a positive and negative attribute depending upon how the iterator is used.

KEY CONCEPT
Level-order traversal means visit the nodes at each level, one level at a time, starting with the root.

9.4 A Binary Tree ADT

Let's take a look at a simple binary tree implementation using links. In Section 9.5, we will consider an example using this implementation. As we discussed earlier in this chapter, it is difficult to abstract an interface for all trees. However, once we have narrowed our focus to binary trees, the task becomes more reasonable. One possible set of operations for a binary tree ADT is listed in Figure 9.8. Keep in mind that the definition of a collection is not universal. You will find variations in the operations defined for specific collections from one book to another. We have been very careful in this book to define the operations on each collection so that they are consistent with its purpose.

Notice that in all of the operations listed, there are no operations to add elements to or remove elements from the tree. This is because until we specify the

Operation	Description
`getRoot`	Returns a reference to the root of the binary tree
`isEmpty`	Determines if the tree is empty
`size`	Returns the number of elements in the tree
`contains`	Determines if the specified target is in the tree
`find`	Returns a reference to the specified target element if it is found
`toString`	Returns a string representation of the tree
`iteratorInOrder`	Returns an iterator for an inorder traversal of the tree
`iteratorPreOrder`	Returns an iterator for a preorder traversal of the tree
`iteratorPostOrder`	Returns an iterator for a postorder traversal of the tree
`iteratorLevelOrder`	Returns an iterator for a level-order traversal of the tree

FIGURE 9.8 The operations on a binary tree

purpose and organization of the binary tree, there is no way to know how or more specifically where to add an element to the tree. Similarly, any operation to remove one or more elements from the tree may violate the purpose or structure of the tree as well. As with adding an element, we do not yet have enough information to know how to remove an element. When we were dealing with stacks in Chapters 3 and 4, we could think about the concept of removing an element from a stack, and it was easy to conceptualize the state of the stack after the removal of the element. The same can be said of queues, because we could remove an element from only one end of the linear structures. Even with lists, where we could remove an element from the middle of the linear structure, it was easy to conceptualize the state of the resulting list.

With a tree, however, upon removing an element, we have many issues to handle that will affect the state of the tree. What happens to the children and other descendants of the element that is removed? Where does the child pointer of the element's parent now point? What if the element we are removing is the root? As we will see in our example using expression trees later in this chapter, there will be applications of trees where there is no concept of the removal of an element from the tree. Once we have specified more detail about the use of the tree, we may then decide that a `removeElement` method is appropriate. An excellent example of this is binary search trees, as we will see in Chapter 10.

Listing 9.1 shows the `BinaryTreeADT` interface. Figure 9.9 shows the UML description for the `BinaryTreeADT` interface.

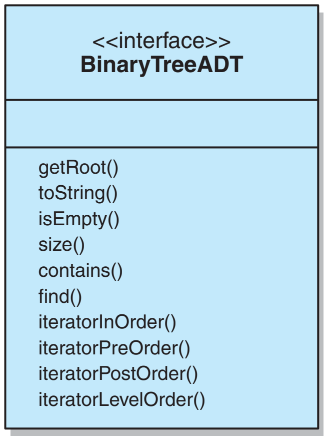


FIGURE 9.9 UML description of the `BinaryTreeADT` interface

LISTING 9.1

```
/**
 * BinaryTreeADT defines the interface to a binary tree data structure.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 8/19/08
 */
package jss2;
import java.util.Iterator;

public interface BinaryTreeADT<T>
{

   /**
    * Returns a reference to the root element
    *
    * @return                a reference to the root
    */
   public T getRoot ();

   /**
    * Returns true if this binary tree is empty and false otherwise.
    *
    * @return  true if this binary tree is empty
    */
   public boolean isEmpty();

   /**
    * Returns the number of elements in this binary tree.
    *
    * @return  the integer number of elements in this tree
    */
   public int size();

   /**
    * Returns true if the binary tree contains an element that matches
    * the specified element and false otherwise.
    *
    * @param targetElement     the element being sought in the tree
    * @return                  true if the tree contains the target element
    */
   public boolean contains (T targetElement);
```

LISTING 9.1 *continued*

```
/**
 * Returns a reference to the specified element if it is found in
 * this binary tree. Throws an exception if the specified element
 * is not found.
 *
 * @param targetElement     the element being sought in the tree
 * @return                  a reference to the specified element
 */
public T find (T targetElement);

/**
 * Returns the string representation of the binary tree.
 *
 * @return a string representation of the binary tree
 */
public String toString();

/**
 * Performs an inorder traversal on this binary tree by calling an
 * overloaded, recursive inorder method that starts with the root.
 *
 * @return an iterator over the elements of this binary tree
 */
public Iterator<T> iteratorInOrder();

/**
 * Performs a preorder traversal on this binary tree by calling an
 * overloaded, recursive preorder method that starts with the root.
 *
 * @return an iterator over the elements of this binary tree
 */
public Iterator<T> iteratorPreOrder();

/**
 * Performs a postorder traversal on this binary tree by calling an
 * overloaded, recursive postorder method that starts with the root.
 *
 * @return an iterator over the elements of this binary tree
 */
public Iterator<T> iteratorPostOrder();
```

LISTING 9.1 *continued*

```
    /**
     * Performs a level-order traversal on the binary tree, using a queue.
     *
     * @return an iterator over the elements of this binary tree
     */
    public Iterator<T> iteratorLevelOrder();
}
```

9.5 Using Binary Trees: Expression Trees

In Chapter 3, we used a stack algorithm to evaluate postfix expressions. In this section, we modify that algorithm to construct an expression tree using an `ExpressionTree` class that extends our definition of a binary tree. Figure 9.10 illustrates the concept of an expression tree. Notice that the root and all of the internal nodes of an expression tree contain operations and that all of the leaves contain operands. An expression tree is evaluated from the bottom up. In this case, the (5–3) term is evaluated first, yielding 2. That result is then multiplied by 4, yielding 8. Finally, the result of that term is added to 9, yielding 17.

Listing 9.2 illustrates our `ExpressionTree` class. This class extends the `LinkedBinaryTree` class, providing a new constructor that will combine expression trees to make a new tree and providing an `evaluate` method to recursively evaluate an expression tree once it has been constructed. Note that this class could have also been written to extend the `ArrayBinaryTree` class, but many of

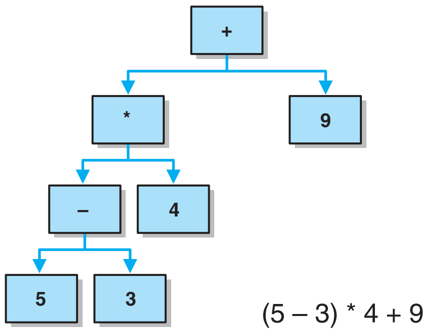


FIGURE 9.10 An example expression tree

the operations would have been significantly different. For example, our constructor that so elegantly combines a new element and two existing trees to form a new tree in constant O(1) time would require the merging of two arrays in the array implementation resulting in at best O(n) time complexity. This does not mean that the linked implementation is always better; it simply means that it fits this particular problem better than the array implementation.

LISTING 9.2

```
/**
 * ExpressionTree represents an expression tree of operators and operands.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 8/19/08
 */
package jss2;

public class ExpressionTree extends LinkedBinaryTree<ExpressionTreeObj>
{

   /**
    * Creates an empty expression tree.
    */
   public ExpressionTree()
   {
      super();
   }

   /**
    * Constructs a expression tree from the two specified expression
    * trees.
    *
    * @param element         the expression tree for the center
    * @param leftSubtree     the expression tree for the left subtree
    * @param rightSubtree    the expression tree for the right subtree
    */
   public ExpressionTree (ExpressionTreeObj element,
               ExpressionTree leftSubtree, ExpressionTree rightSubtree)
   {
      root = new BinaryTreeNode<T> (element);
      count = 1;
```

LISTING 9.2 *continued*

```
      if (leftSubtree != null)
      {
         count = count + leftSubtree.size();
         root.left = leftSubtree.root;
      }
      else
         root.left = null;

      if (rightSubtree !=null)
      {
         count = count + rightSubtree.size();
         root.right = rightSubtree.root;
      }
      else
         root.right = null;
   }

   /**
    * Evaluates the expression tree by calling the recursive
    * evaluateNode method.
    *
    * @return  the integer evaluation of the tree
    */
   public int evaluateTree()
   {
      return evaluateNode(root);
   }

   /**
    * Recursively evaluates each node of the tree.
    *
    * @param root   the root of the tree to be evaluated
    * @return       the integer evaluation of the tree
    */
   public int evaluateNode(BinaryTreeNode root)
   {
      int result, operand1, operand2;
      ExpressionTreeObj temp;

      if (root==null)
         result = 0;
      else
      {
```

LISTING 9.2 *continued*

```
         temp = (ExpressionTreeObj)root.element;

         if (temp.isOperator())
         {
            operand1 = evaluateNode(root.left);
            operand2 = evaluateNode(root.right);
            result = computeTerm(temp.getOperator(), operand1, operand2);
         }
         else
            result = temp.getValue();
      }

      return result;
   }

   /**
    * Evaluates a term consisting of an operator and two operands.
    *
    * @param operator   the operator for the expression
    * @param operand1   the first operand for the expression
    * @param operand2   the second operand for the expression
    */
   private static int computeTerm(char operator, int operand1, int operand2)
   {
      int result=0;

      if (operator == '+')
         result = operand1 + operand2;

      else if (operator == '-')
         result = operand1 - operand2;
      else if (operator == '*')
         result = operand1 * operand2;
      else
         result = operand1 / operand2;

      return result;
   }
}
```

The evaluateTree method calls the recursive evaluateNode method. The evaluateNode method returns the value if the node contains a number, or it returns the result of the operation using the value of the left and right subtrees if the node contains an operation. The ExpressionTree class uses the ExpressionTreeObj class as the element to store at each node of the tree. The ExpressionTreeObj class allows us to keep track of whether the element is a number or an operator and which operator or what value is stored there. The ExpressionTreeObj class is illustrated in Listing 9.3.

LISTING 9.3

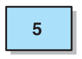
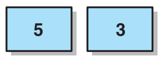
```
/**
 * ExpressionTreeObj represents an element in an expression tree.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 8/19/08
 */
package jss2;

public class ExpressionTreeObj
{

   private int termType;
   private char operator;
   private int value;

   /**
    * Creates a new expression tree object with the specified data.
    *
    * @param type   the integer type of the expression
    * @param op     the operand for the expression
    * @param val    the value for the expression
    */
   public ExpressionTreeObj (int type, char op, int val)
   {
      termType = type;
      operator = op;
      value = val;
   }
```

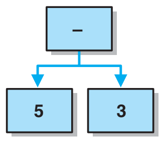
LISTING 9.3 *continued*

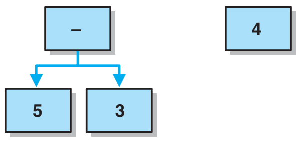
```
   /**
    * Returns true if this object is an operator and false otherwise.
    *
    * @return true if this object is an operator
    */
   public boolean isOperator()
   {
      return (termType == 1);
   }
   /**
    * Returns the operator of this expression tree object.
    *
    * @return the character representation of the operator
    */
   public char getOperator()
   {
      return operator;
   }

   /**
    * Returns the value of this expression tree object.
    *
    * @return the value of this expression tree object
    */
   public int getValue()
   {
      return value;
   }
}
```

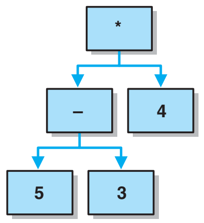
The `Postfix` and `PostfixEvaluator` classes are a modification of our solution from Chapter 3. This solution allows the user to enter a postfix expression from the keyboard. As each term is entered, if it is an operand, a new `ExpressionTreeObj` is created with the given value and then an `ExpressionTree` is constructed using that element as the root and with no children. The new `ExpressionTree` is then pushed onto a stack. If the term entered is an operator, the top two `ExpressionTrees` on the stack are popped off, a new `ExpressionTreeObj` is created with the given operator value, and a new `ExpressionTree` is created with this operator as the root and the two `ExpressionTrees` popped off of the stack as the left and right subtrees. Figure 9.11 illustrates this process for the expression tree from Figure 9.10. Note that the top of the expression tree stack is on the right.

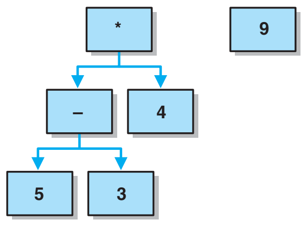
Input in Postfix: 5 3 – 4 * 9 +

Token	Processing Steps	Expression Tree Stack (top at right)
5	push(new ExpressionTree(5, null, null))	
3	push(new ExpressionTree(3, null, null))	

–	op2 = pop
	op1 = pop
	push(new ExpressionTree(–, op1, op2))
		

4	push(new ExpressionTree(4, null, null))	

*	op2 = pop
	op1 = pop
	push(new ExpressionTree(*, op1, op2))
		

9	push(new ExpressionTree(9, null, null))	

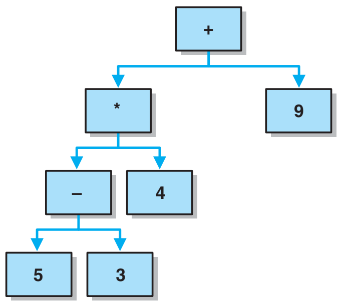
+	op2 = pop
	op1 = pop
	push(new ExpressionTree(+, op1, op2))
		

FIGURE 9.11 Building an Expression Tree from a postfix expression

LISTING 9.4

```
/**
 * Postfix2 uses the PostfixEvaluator class to solve a postfix expression
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 8/19/08
 */
public class Postfix2
{

   /**
    * Uses the PostfixEvaluator class to solve a postfix expression.
    */
   public static void main (String[] args)
   {
      PostfixEvaluator2 temp = new PostfixEvaluator2();
      temp.solve();
   }
}
```

The `Postfix` class is shown in Listing 9.4, and the `PostfixEvaluator` class is shown in Listing 9.5. The UML description of the `Postfix` class is shown in Figure 9.12.

9.6 Implementing Binary Trees with Links

We will examine how some of these methods might be implemented using a linked implementation, while others will be left as exercises. The `LinkedBinaryTree` class implementing the `BinaryTreeADT` interface will need to keep track of the node that is at the root of the tree and the number of elements on the tree. The `LinkedBinaryTree` instance data could be declared as

```
protected int count;
protected BinaryTreeNode<T> root;
```

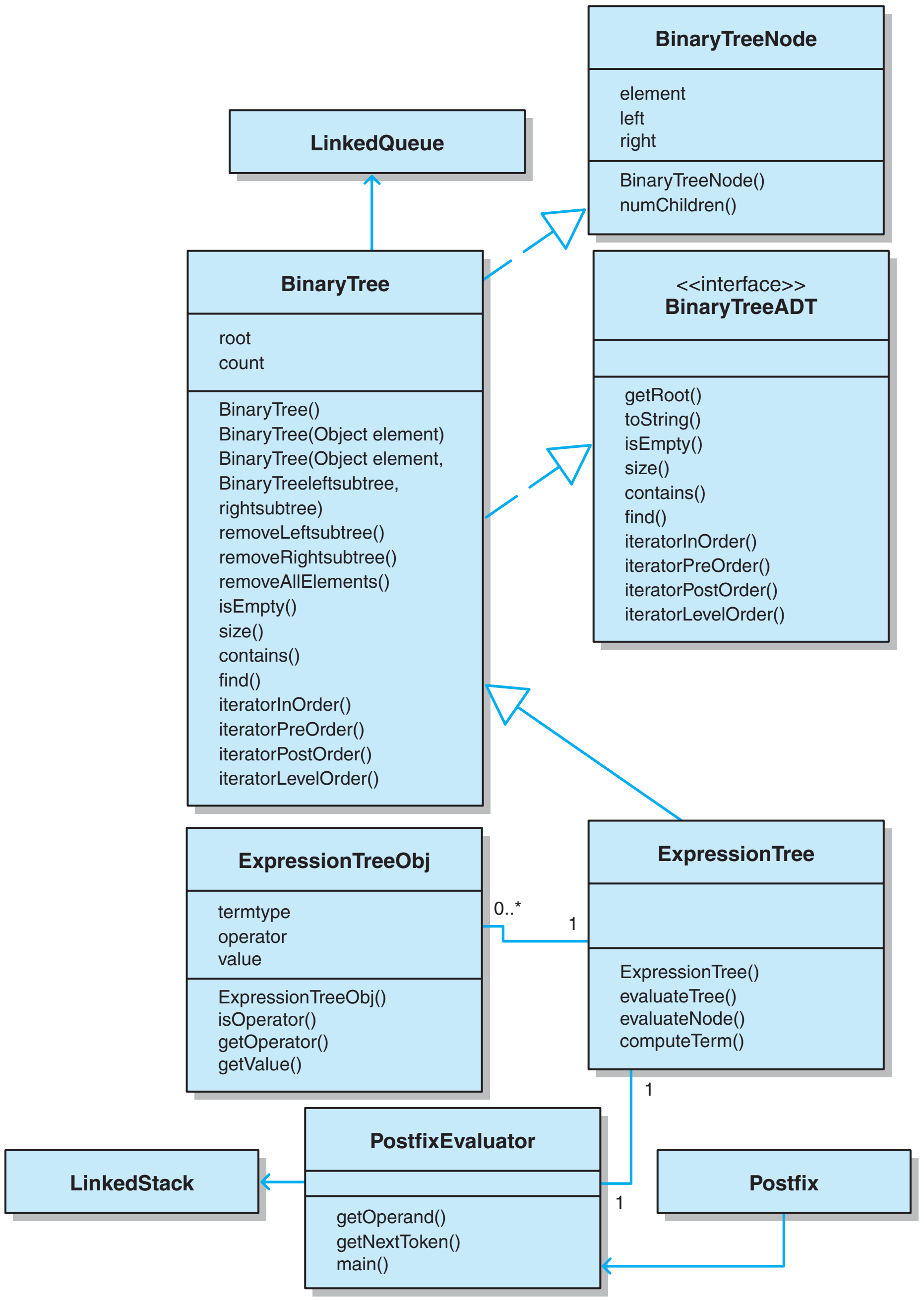


FIGURE 9.12 UML description of the Postfix example

LISTING 9.5

```java
/**
 * PostfixEvaluator2: this modification of our stack example uses a pair of
 * stacks to create an expression tree from a VALID postfix integer expression
 * and then uses a recursive method from the ExpressionTree class to
 * evaluate the tree.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 8/19/08
 */
import jss2.*;
import jss2.exceptions.*;
import java.util.StringTokenizer;
import java.util.Iterator;
import java.io.*;

public class PostfixEvaluator2
{
   /**
    * Retrieves and returns the next operand off of this tree stack.
    *
    * @param treeStack   the tree stack from which the operand will be returned
    * @return            the next operand off of this tree stack
    */
   private ExpressionTree getOperand(LinkedStack<ExpressionTree> treeStack)
   {
      ExpressionTree temp;
      temp = treeStack.pop();

      return temp;
   }

   /**
    * Retrieves and returns the next token, either an operator or
    * operand from the user.
    *
    * @return the next string token
    */
   private String getNextToken()
   {
      String tempToken = "0", inString;
      StringTokenizer tokenizer;
```

LISTING 9.5 *continued*

```
      try
      {
         BufferedReader in =
               new BufferedReader(new InputStreamReader(System.in));
         inString = in.readLine();
         tokenizer = new StringTokenizer(inString);
         tempToken = (tokenizer.nextToken());
      }
      catch (Exception IOException)
      {
         System.out.println("An input/output exception has occurred");
      }

      return tempToken;
   }

   /**
    * Prompts the user for a valid post-fix expression, converts it to
    * an expression tree using a two stack method, then calls a
    * recursive method to evaluate the expression.
    */
   public void solve ()
   {
      ExpressionTree operand1, operand2;
      char operator;
      String tempToken;
      LinkedStack<ExpressionTree> treeStack = new LinkedStack<ExpressionTree>();

      System.out.println("Enter a valid post-fix expression one token " +
                         "at a time pressing the enter key after each token");
      System.out.println("Enter an integer, an operator (+,-,*,/)" +
                         "then ! to evaluate");

      tempToken = getNextToken();
      operator = tempToken.charAt(0);

      while (!(operator == '!'))
      {
         if ((operator == '+') || (operator == '-') || (operator == '*') ||
             (operator == '/'))
         {
```

LISTING 9.5 *continued*

```
               operand1 = getOperand(treeStack);
               operand2 = getOperand(treeStack);
               treeStack.push(new ExpressionTree
                        (new ExpressionTreeObj(1,operator,0), operand2, operand1));
            }
         else
            {
               treeStack.push(new ExpressionTree (new ExpressionTreeObj
                             (2,' ', Integer.parseInt(tempToken)), null, null));
            }

            tempToken = getNextToken();
            operator = tempToken.charAt(0);
         }

         System.out.print("The result is ");
         System.out.println(((ExpressionTree)treeStack.peek()).evaluateTree());
      }
   }
```

The constructors for the `LinkedBinaryTree` class should handle two cases: We want to create a binary tree with nothing in it, and we want to create a binary tree with a single element but no children. Neither of these possibilities should violate any specific organization of a binary tree. With these goals in mind, the `LinkedBinaryTree` class might have the following constructors. Note that each of the constructors must account for both the `root` and `count` attributes.

```
   /**
    * Creates an empty binary tree.
    */
   public LinkedBinaryTree()
   {
      count = 0;
      root = null;
   }
   /**
    * Creates a binary tree with the specified element as its root.
    *
    * @param element the element that will become the root of the new binary
    *                tree
    */
   public LinkedBinaryTree (T element)
   {
      count = 1;
      root = new BinaryTreeNode<T> (element);
   }
```

Note that both the instance data and the constructors use an additional class called `BinaryTreeNode`. As discussed earlier, this class keeps track of the element stored at each location as well as pointers to the left and right subtree or children of each node. In this particular implementation, we chose not to include a pointer back to the parent of each node. Listing 9.6 shows the `BinaryTreeNode` class. The `BinaryTreeNode` class also includes a recursive method to return the number of children of the given node.

There are a variety of other possibilities for implementation of a tree node or binary tree node class. For example, methods could be included to test whether

LISTING 9.6

```
/**
 * BinaryTreeNode represents a node in a binary tree with a left and
 * right child.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 8/19/08
 */
package jss2;

public class BinaryTreeNode<T>
{
   protected T element;
   protected BinaryTreeNode<T> left, right;
```

LISTING 9.6 *continued*

```
   /**
    * Creates a new tree node with the specified data.
    *
    * @param obj the element that will become a part of the new tree node
    */
   BinaryTreeNode (T obj)
   {
      element = obj;
      left = null;
      right = null;
   }

   /**
    * Returns the number of non-null children of this node.
    * This method may be able to be written more efficiently.
     *
    * @return the integer number of non-null children of this node
    */
   public int numChildren()
   {
      int children = 0;

      if (left != null)
         children = 1 + left.numChildren();

      if (right != null)
         children = children + 1 + right.numChildren();

      return children;
   }
}
```

the node is a leaf (i.e., does not have any children), to test whether the node is an internal node (i.e., has at least one child), to test the depth of the node from the root, or to calculate the height of the left and right subtrees.

Another alternative would be to use polymorphism such that, rather than testing a node to see if it has data or has children, we would create various implementations, such as an `emptyTreeNode`, an `innerTreeNode`, and a `leafTreeNode`, that would distinguish the various possibilities.

The `find` Method

As with our earlier collections, our `find` method traverses the tree using the `equals` method of the class stored in the tree to determine equality. This puts the definition of equality under the control of the class being stored in the tree. The `find` method throws an exception if the target element is not found.

Many methods associated with trees may be written either recursively or iteratively. Often, when written recursively, these methods require the use of a private support method because the signature and/or the behavior of the first call and each successive call may not be the same. The `find` method in our simple implementation is an excellent example of this strategy.

We have chosen to use a recursive `findAgain` method. We know that the first call to `find` will start at the root of the tree, and if that instance of the `find` method completes without finding the target, we need to throw an exception. The private `findAgain` method allows us to distinguish between this first instance of the `find` method and each successive call.

```
/**
 * Returns a reference to the specified target element if it is
 * found in this binary tree. Throws a NoSuchElementException if
 * the specified target element is not found in the binary tree.
 *
 * @param targetElement               the element being sought in this tree
 * @return                            a reference to the specified target
 * @throws ElementNotFoundException   if an element not found exception
occurs
 */
public T find(T targetElement) throws ElementNotFoundException
{
   BinaryTreeNode<T> current = findAgain(targetElement, root);

   if( current == null )
      throw new ElementNotFoundException("binary tree");

   return (current.element);
}

/**
 * Returns a reference to the specified target element if it is
 * found in this binary tree.
 *
```

```
     * @param targetElement      the element being sought in this tree
     * @param next               the element to begin searching from
     */
    private BinaryTreeNode<T> findAgain(T targetElement,
                                        BinaryTreeNode<T> next)
    {
       if (next == null)
          return null;

       if (next.element.equals(targetElement))
          return next;

       BinaryTreeNode<T> temp = findAgain(targetElement, next.left);

       if (temp == null)
          temp = findAgain(targetElement, next.right);
       return temp;
    }
```

As seen in earlier examples, the `contains` method can make use of the find method. Our implementation of this is left as a programming project.

The `iteratorInOrder` Method

Another interesting operation is the `iteratorInOrder` method. The task is to create an `Iterator` object that will allow a user class to step through the elements of the tree in an inorder traversal. The solution to this problem provides another example of using one collection to build another. We simply traverse the tree using a definition of "visit" from earlier pseudocode that adds the contents of the node onto an unordered list. We then return the list iterator as the result of the `iterator` method for our tree. This approach is possible because of the linear nature of an unordered list and the way that we implemented the iterator method for a list. The iterator method for a list returns a `LinkedIterator` that starts with the element at the front of the list and steps through the list in a linear fashion. It is important to understand that this behavior is not a requirement for an iterator associated with a list. It is simply an artifact of the way that we chose to implement the `iterator` method for a list and the `LinkedIterator` class.

Like the `find` operation, we use a private helper method in our recursion.

```
/**
 * Performs an inorder traversal on this binary tree by calling an
 * overloaded, recursive inorder method that starts with
 * the root.
 *
 * @return an in order iterator over this binary tree
 */
public Iterator<T> iteratorInOrder()
{
   ArrayUnorderedList<T> tempList = new ArrayUnorderedList<T>();
   inorder (root, tempList);

   return tempList.iterator();
}

/**
 * Performs a recursive inorder traversal.
 *
 * @param node      the node to be used as the root for this traversal
 * @param tempList  the temporary list for use in this traversal
 */
protected void inorder (BinaryTreeNode<T> node,
                        ArrayUnorderedList<T> tempList)
{
   if (node != null)
   {
      inorder (node.left, tempList);
      tempList.addToRear(node.element);
      inorder (node.right, tempList);
   }
}
```

The other iterator operations are similar and are left as exercises.

9.7 Implementing Binary Trees with Arrays

Now let us consider how some of these methods might be implemented using an array implementation. Again, many of the methods will be left as exercises. The `ArrayBinaryTree` class implementing the `BinaryTreeADT` interface will use the computational method we described earlier. Thus the left child of a node stored at position n will be stored in position (2 * n + 1), and that element's right child will be stored in position (2 * (n + 1)).

KEY CONCEPT

In the computational strategy to implement a tree with an array, the children of node n are stored at 2n + 1 and 2(n + 1), respectively.

The ArrayBinaryTree instance data could be declared as

```
protected int count;
protected T[] tree;
```

Like the constructors for the LinkedBinaryTree class, the constructors for the ArrayBinaryTree class should handle two cases: We want to create a binary tree with nothing in it, and we want to create a binary tree with a single element but no children. Neither of these possibilities should violate any specific organization of a binary tree.

```
/**
 * Creates an empty binary tree.
 */
public ArrayBinaryTree()
{
   count = 0;
   tree = (T[]) new Object[capacity];
}

/**
 * Creates a binary tree with the specified element as its root.
 *
 * @param element  the element which will become the root of the new tree
 */
public ArrayBinaryTree (T element)
{
   count = 1;
   tree = (T[]) new Object[capacity];
   tree[0] = element;
}
```

Note that unlike our linked implementation, there is no need to use the BinaryTreeNode class. We can simply store the elements directly in the array. However, we will need a method to expand the capacity of the array just as we did with our earlier array implementations.

```
/**
 * Private method to expand capacity if full.
 */
```

```
protected void expandCapacity()
{
   T[] temp = (T[]) new Object[tree.length * 2];

   for (int ct=0; ct < tree.length; ct++)
      temp[ct] = tree[ct];

   tree = temp;
}
```

The `find` Method

As with our earlier collections, our `find` method traverses the tree using the `equals` method of the class stored in the tree to determine equality. This puts the definition of equality under the control of the class being stored in the tree. The `find` method throws an exception if the target element is not found.

Because we are using an array implementation of a tree, one seemingly brute force solution to finding an element in the tree would be to simply use a linear search of the array. Keep in mind that we do not know anything about the ordering of the elements in our tree. Therefore, the element we are searching for may be in any location of our array.

We could attempt to use a recursive solution like the one we used for the linked implementation. However, given our array implementation, we do not need to incur the overhead of a recursive algorithm. We will, instead, use a simple, O(n), linear search of the array. Keep in mind that as we extend our tree implementations in later chapters to add order, we will be able to improve upon the time complexity of this operation.

```
/**
 * Returns a reference to the specified target element if it is
 * found in this binary tree. Throws a NoSuchElementException if
 * the specified target element is not found in the binary tree.
 *
 * @param targetElement                the element being sought in the tree
 * @return                             true if the element is in the tree
 * @throws ElementNotFoundException    if an element not found exception occurs
 */
```

```
public T find (T targetElement) throws ElementNotFoundException
{
   T temp=null;
   boolean found = false;

   for (int ct=0; ct<count && !found; ct++)
      if (targetElement.equals(tree[ct]))
      {
         found = true;
         temp = tree[ct];
      }
   if (!found)
      throw new ElementNotFoundException("binary tree");

   return temp;
}
```

The `contains` method, as we did in earlier examples, can make use of the `find` method and is left as a programming project.

The `iteratorInOrder` Method

Another interesting operation is the `iteratorInOrder` method. As with the linked implementation, the task is to create an `iterator` object that will allow a user class to step through the elements of the tree in an inorder traversal. The solution to this problem provides another example of using one collection to build another. It is identical to the method we used in the linked implementation. We simply traverse the tree using a definition of “visit” from earlier pseudocode that adds the contents of the node onto an unordered list. We then return the list iterator as the result of the `iterator` method for our tree.

Like the `find` operation, we use a private helper method in our recursion.

```
/**
 * Performs an inorder traversal on this binary tree by calling an
 * overloaded, recursive inorder method that starts with
 * the root.
 *
 * @return  an iterator over the binary tree
 */
```

```
public Iterator<T> iteratorInOrder()
{
   ArrayUnorderedList<T> templist = new ArrayUnorderedList<T>();
   inorder (0, templist);

   return templist.iterator();
}

/**
 * Performs a recursive inorder traversal.
 *
 * @param node      the node used in the traversal
 * @param templist  the temporary list used in the traversal
 */
protected void inorder (int node, ArrayUnorderedList<T> templist)
{
   if (node < tree.length)
      if (tree[node] != null)
   {
      inorder (node*2+1, templist);
      templist.addToRear(tree[node]);
      inorder ((node+1)*2, templist);
   }
}
```

The other iterator operations are similar and are left as exercises.

Summary of Key Concepts

- A tree is a nonlinear structure whose elements are organized into a hierarchy.
- Trees are described by a large set of related terms.
- The simulated link strategy allows array positions to be allocated contiguously regardless of the completeness of the tree.
- In general, a balanced n-ary tree with m elements will have height $\log_n m$.
- There are four basic methods for traversing a tree: preorder, inorder, postorder, and level-order.
- Preorder traversal means visit the node, then the left child, then the right child.
- Inorder traversal means visit the left child, then the node, then the right child.
- Postorder traversal means visit the left child, then the right child, then the node.
- Level-order traversal means visit the nodes at each level, one level at a time, starting with the root.
- In the computational strategy to implement a tree with an array, the children of node n are stored at $2n + 1$ and $2(n + 1)$, respectively.

Self-Review Questions

SR 9.1 What is a tree?

SR 9.2 What is a node?

SR 9.3 What is the root of the tree?

SR 9.4 What is a leaf?

SR 9.5 What is an internal node?

SR 9.6 Define the height of a tree.

SR 9.7 Define the level of a node.

SR 9.8 What are the advantages and disadvantages of the computational strategy?

SR 9.9 What are the advantages and disadvantages of the simulated link strategy?

SR 9.10 What attributes should be stored in the `TreeNode` class?

SR 9.11 Which method of traversing a tree would result in a sorted list for a binary search tree?

SR 9.12 We used a list to implement the iterator methods for a binary tree. What must be true for this strategy to be successful?

Exercises

EX 9.1 Develop a pseudocode algorithm for a level-order traversal of a binary tree.

EX 9.2 Draw either a matrilineage (following your mother's lineage) or a patrilineage (following your father's lineage) diagram for a couple of generations. Develop a pseudocode algorithm for inserting a person into their proper place in the tree.

EX 9.3 Develop a pseudocode algorithm to build an expression tree from a prefix expression.

EX 9.4 Develop a pseudocode algorithm to build an expression tree from an infix expression.

EX 9.5 Calculate the time complexity of the `find` method.

EX 9.6 Calculate the time complexity of the `iteratorInOrder` method.

EX 9.7 Develop a pseudocode algorithm for the `size` method assuming that there is not a `count` variable.

EX 9.8 Develop a pseudocode algorithm for the `isEmpty` operation assuming that there is not a `count` variable.

EX 9.9 Draw an expression tree for the expression $(9 + 4) * 5 + (4 - (6 - 3))$.

Programming Projects

PP 9.1 Complete the implementation of the `getRoot` and `toString` operations of a binary tree.

PP 9.2 Complete the implementation of the `size` and `isEmpty` operations of a binary tree, assuming that there is not a `count` variable.

PP 9.3 Create boolean methods for our `BinaryTreeNode` class to determine if the node is a leaf or an internal node.

PP 9.4 Create a method called `depth` that will return an `int` representing the level or depth of the given node from the root.

PP 9.5 Complete the implementation of the `contains` method for a binary tree.

PP 9.6 Implement the `contains` method for a binary tree without using the `find` operation.

PP 9.7 Complete the implementation of the iterator methods for a binary tree.

PP 9.8 Implement the iterator methods for a binary tree without using a list.

PP 9.9 Modify the `ExpressionTree` class to create a method called `draw` that will graphically depict the expression tree.

PP 9.10 We use postfix notation in the example in this chapter because it eliminates the need to parse an infix expression by precedence rules and parentheses. Some infix expressions do not need parentheses to modify precedence. Implement a method for the `ExpressionTree` class that will determine if an integer expression would require parentheses if it were written in infix notation.

PP 9.11 Create an array-based implementation of a binary tree using the computational strategy.

PP 9.12 Create an array-based implementation of a binary tree using the simulated link strategy.

Answers to Self-Review Questions

SRA 9.1 A tree is a nonlinear structure defined by the concept that each node in the tree, other than the first node or root node, has exactly one parent.

SRA 9.2 Node refers to a location in the tree where an element is stored.

SRA 9.3 Root refers to the node at the base of the tree or the one node in the tree that does not have a parent.

SRA 9.4 A leaf is a node that does not have any children.

SRA 9.5 An internal node is any non-root node that has at least one child.

SRA 9.6 The height of the tree is the length of the longest path from the root to a leaf.

SRA 9.7 The level of a node is measured by the number of links that must be followed to reach that node from the root.

SRA 9.8 The computational strategy does not have to store links from parent to child because that relationship is fixed by position.

However, this strategy may lead to substantial wasted space for trees that are not balanced and/or not complete.

SRA 9.9 The simulated link strategy stores array index values as pointers between parent and child and allows the data to be stored contiguously no matter how balanced and/or complete the tree. However, this strategy increases the overhead in terms of maintaining a freelist or shifting elements in the array.

SRA 9.10 The `TreeNode` class must store a pointer to the element stored in that position as well as pointers to each of the children of that node. The class may also contain a pointer to the parent of the node.

SRA 9.11 Inorder traversal of a binary search tree would result in a sorted list in ascending order.

SRA 9.12 For this strategy to be successful, the iterator for a list must return the elements in the order in which they were added. For this particular implementation of a list, we know this is indeed the case.

This page intentionally left blank

10 Binary Search Trees

In this chapter, we will explore the concept of binary search trees and options for their implementation. We will examine algorithms for adding and removing elements from binary search trees and for maintaining balanced binary search trees. We will discuss the analysis of these implementations and also explore various uses of binary search trees.

CHAPTER OBJECTIVES

- Define a binary search tree abstract data structure
- Demonstrate how a binary search tree can be used to solve problems
- Examine various binary search tree implementations
- Compare binary search tree implementations

10.1 A Binary Search Tree

A *binary search tree* is a binary tree with the added property that, for each node, the left child is less than the parent, which is less than or equal to the right child. As we discussed in Chapter 9, it is very difficult to abstract a set of operations for a tree without knowing what type of tree it is and its intended purpose. With the added ordering property that must be maintained, we can now extend our definition to include the operations on a binary search tree listed in Figure 10.1.

KEY CONCEPT

A binary search tree is a binary tree with the added property that the left child is less than the parent, which is less than or equal to the right child.

KEY CONCEPT

The definition of a binary search tree is an extension of the definition of a binary tree.

We must keep in mind that the definition of a binary search tree is an extension of the definition of a binary tree discussed in the last chapter. Thus, these operations are in addition to the ones defined for a binary tree. Keep in mind that at this point we are simply discussing binary search trees, but as we will see shortly, the interface for a balanced binary search tree will be the same. Listing 10.1 and Figure 10.2 describe a `BinarySearchTreeADT`.

LISTING 10.1

```
/**
 * BinarySearchTreeADT defines the interface to a binary search tree.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 8/19/08
 */
package jss2;

public interface BinarySearchTreeADT<T> extends BinaryTreeADT<T>
{

   /**
    * Adds the specified element to the proper location in this tree.
    *
    * @param element the element to be added to this tree
    */

   public void addElement (T element);
```

LISTING 10.1 *continued*

```
/**
 * Removes and returns the specified element from this tree.
 *
 * @param targetElement  the element to be removed from this tree
 * @return               the element removed from this tree
 */

public T removeElement (T targetElement);

/**
 * Removes all occurrences of the specified element from this tree.
 *
 * @param targetElement  the element that the list will have all instances
 *                       of it removed
 */

public void removeAllOccurrences (T targetElement);

/**
 * Removes and returns the smallest element from this tree.
 *
 * @return the smallest element from this tree.
 */

public T removeMin();

/**
 * Removes and returns the largest element from this tree.
 *
 * @return the largest element from this tree
 */

public T removeMax();

/**
 * Returns a reference to the smallest element in this tree.
 *
 * @return a reference to the smallest element in this tree
 */

public T findMin();
```

LISTING 10.1 *continued*

```
    /**
     * Returns a reference to the largest element in this tree.
     *
     * @return a reference to the largest element in this tree
     */

    public T findMax();
}
```

10.2 Implementing Binary Search Trees: With Links

In Chapter 9, we introduced a simple implementation of a `LinkedBinaryTree` class using a `BinaryTreeNode` class to represent each node of the tree. Each `BinaryTreeNode` object maintains a reference to the element stored at that node as well as references to each of the node's children. We can simply extend that definition with a `LinkedBinarySearchTree` class implementing the `BinarySearchTreeADT` interface. Because we are extending the `LinkedBinaryTree` class from Chapter 9, all of the methods we discussed are still supported, including the various traversals.

KEY CONCEPT

Each `BinaryTreeNode` object maintains a reference to the element stored at that node as well as references to each of the node's children.

Our `LinkedBinarySearchTree` class offers two constructors: one to create an empty `LinkedBinarySearchTree` and the other to

Operation	Description
`addElement`	Add an element to the tree.
`removeElement`	Remove an element from the tree.
`removeAllOccurrences`	Remove all occurrences of element from the tree.
`removeMin`	Remove the minimum element in the tree.
`removeMax`	Remove the maximum element in the tree.
`findMin`	Returns a reference to the minimum element in the tree.
`findMax`	Returns a reference to the maximum element in the tree.

FIGURE 10.1 The operations on a binary search tree

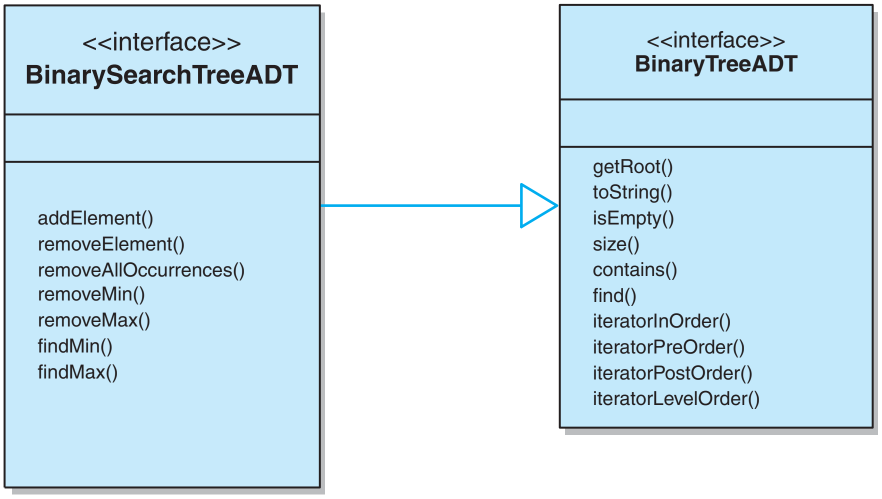


FIGURE 10.2 UML description of the `BinarySearchTreeADT`

create a `LinkedBinarySearchTree` with a particular element at the root. Both of these constructors simply refer to the equivalent constructors of the super class (i.e., the `LinkedBinaryTree` class).

```
/**
 * Creates an empty binary search tree.
 */
public LinkedBinarySearchTree()
{
   super();
}

/**
 * Creates a binary search with the specified element as its root.
 *
 * @param element  the element that will be the root of the new binary
 *                 search tree
 */
public LinkedBinarySearchTree (T element)
{
   super (element);
}
```

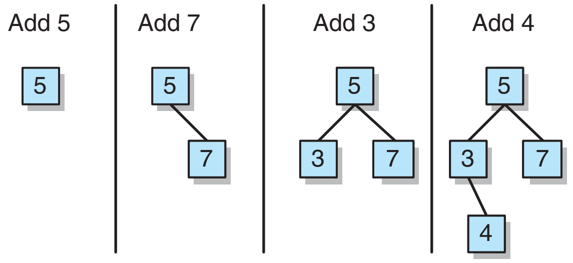


FIGURE 10.3 Adding elements to a binary search tree

The `addElement` Operation

The `addElement` method adds a given element to an appropriate location in the tree, given its value. If the element is not `Comparable`, the method throws a `ClassCastException`. If the tree is empty, the new element becomes the root. If the tree is not empty, the new element is compared to the element at the root. If it is less than the element stored at the root and the left child of the root is null, then the new element becomes the left child of the root. If the new element is less than the element stored at the root and the left child of the root is not null, then we traverse to the left child of the root and compare again. If the new element is greater than or equal to the element stored at the root and the right child of the root is null, then the new element becomes the right child of the root. If the new element is greater than or equal to the element stored at the root and the right child of the root is not null, then we traverse to the right child of the root and compare again. Figure 10.3 illustrates this process of adding elements to a binary search tree. Note that we have chosen to implement this method iteratively. However, this method could have very naturally and elegantly been implemented recursively. Creating the recursive implementation is left as a programming project.

DESIGN FOCUS

Once we have a definition of the type of tree that we wish to construct and how it is to be used, we have the ability to define an interface and implementations. In Chapter 9, we defined a binary tree that enabled us to define a very basic set of operations. Now that we have limited our scope to a binary search tree, we can fill in more details of the interface and the implementation. Determining the level at which to build interface descriptions and determining the boundaries between parent and child classes are design choices . . . and not always easy design choices.

```
/**
 * Adds the specified object to the binary search tree in the
 * appropriate position according to its key value. Note that
 * equal elements are added to the right.
 *
 * @param element the element to be added to the binary search tree
 */
public void addElement (T element)
{
   BinaryTreeNode<T> temp = new BinaryTreeNode<T> (element);
   Comparable<T> comparableElement = (Comparable<T>)element;

   if (isEmpty())
      root = temp;
   else
   {
      BinaryTreeNode<T> current = root;
      boolean added = false;

      while (!added)
      {
         if (comparableElement.compareTo(current.element) < 0)
         {
            if (current.left == null)
            {
               current.left = temp;
               added = true;
            }
            else
               current = current.left;
         }
         else
         {
            if (current.right == null)
            {
               current.right = temp;
               added = true;
            }
            else
               current = current.right;
         }
      }
   }
   count++;
}
```

The `removeElement` Operation

KEY CONCEPT
In removing an element from a binary search tree, another node must be promoted to replace the node being removed.

The `removeElement` method removes a given `Comparable` element from a binary search tree or throws an `ElementNotFoundException` if the given target is not found in the tree. Unlike our earlier study of linear structures, we cannot simply remove the node by making the reference point around the node to be removed. Instead, another node will have to be *promoted* to replace the one being removed. The protected method `replacement` returns a reference to a node that will replace the one specified for removal. There are three cases for selecting the replacement node:

- If the node has no children, `replacement` returns null.
- If the node has only one child, `replacement` returns that child.
- If the node to be removed has two children, `replacement` returns the inorder successor of the node to be removed (because equal elements are placed to the right).

```
/**
 * Removes the first element that matches the specified target
 * element from the binary search tree and returns a reference to
 * it.  Throws a ElementNotFoundException if the specified target
 * element is not found in the binary search tree.
 *
 * @param targetElement              the element being sought in the binary
 *                                   search tree
 * @throws ElementNotFoundException  if an element not found exception occurs
 */
public T removeElement (T targetElement)
                       throws ElementNotFoundException
{
   T result = null;

   if (!isEmpty())
   {
      if (((Comparable)targetElement).equals(root.element))
      {
         result = root.element;
         root = replacement (root);
         count--;
      }
```

```
         else
         {
            BinaryTreeNode<T> current, parent = root;
            boolean found = false;

            if (((Comparable)targetElement).compareTo(root.element) < 0)
               current = root.left;
            else
               current = root.right;

            while (current != null && !found)
            {
               if (targetElement.equals(current.element))
               {
                  found = true;
                  count--;
                  result = current.element;

                  if (current == parent.left)
                  {
                     parent.left = replacement (current);
                  }
                  else
                  {
                     parent.right = replacement (current);
                  }
               }
               else
               {
                  parent = current;

                  if (((Comparable)targetElement).compareTo(current.element) < 0)
                     current = current.left;
                  else
                     current = current.right;
               }
            } // end while

            if (!found)
               throw new ElementNotFoundException("binary search tree");
         }
      } // end outer if

      return result;
   }
```

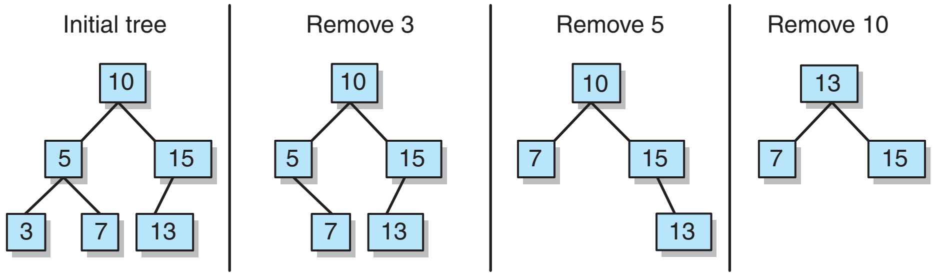


FIGURE 10.4 Removing elements from a binary tree

The following code illustrates the `replacement` method. Figure 10.4 further illustrates the process of removing elements from a binary search tree.

```
/**
 * Returns a reference to a node that will replace the one
 * specified for removal.  In the case where the removed node has
 * two children, the inorder successor is used as its replacement.
 *
 * @param node  the node to be removed
 * @return      a reference to the replacing node
 */
protected BinaryTreeNode<T> replacement (BinaryTreeNode<T> node)
{
   BinaryTreeNode<T> result = null;

   if ((node.left == null)&&(node.right==null))
      result = null;

   else if ((node.left != null)&&(node.right==null))
      result = node.left;

   else if ((node.left == null)&&(node.right != null))
      result = node.right;

   else
   {
      BinaryTreeNode<T> current = node.right;
      BinaryTreeNode<T> parent = node;
```

```
         while (current.left != null)
         {
            parent = current;
            current = current.left;
         }

         if (node.right == current)
            current.left = node.left;

         else
         {
            parent.left = current.right;
            current.right = node.right;
            current.left = node.left;
         }
         result = current;
      }

      return result;
   }
```

The removeAllOccurrences Operation

The `removeAllOccurrences` method removes all occurrences of a given element from a binary search tree and throws an `ElementNotFoundException` if the given element is not found in the tree. This method also throws a `ClassCastException` if the element given is not `Comparable`. This method makes use of the `removeElement` method by calling it once, guaranteeing that the exception will be thrown if there is not at least one occurrence of the element in the tree. The `removeElement` method is then called again as long as the tree contains the target element.

```
   /**
    * Removes elements that match the specified target element from
    * the binary search tree.  Throws a ElementNotFoundException if
    * the specified target element is not found in this tree.
    *
    * @param targetElement              the element being sought in the binary
    *                                   search tree
    * @throws ElementNotFoundException  if an element not found exception occurs
    */
   public void removeAllOccurrences (T targetElement)
                                     throws ElementNotFoundException
   {
      removeElement(targetElement);

      try
      {
         while (contains( (T) targetElement))
            removeElement(targetElement);
      }

      catch (Exception ElementNotFoundException)
      {
      }
   }
```

The `removeMin` Operation

There are three possible cases for the location of the minimum element in a binary search tree:

- If the root has no left child, then the root is the minimum element and the right child of the root becomes the new root.
- If the leftmost node of the tree is a leaf, then it is the minimum element and we simply set its parent's left child reference to null.
- If the leftmost node of the tree is an internal node, then we set its parent's left child reference to point to the right child of the node to be removed.

KEY CONCEPT
The leftmost node in a binary search tree will contain the minimum element whereas the rightmost node will contain the maximum element.

Figure 10.5 illustrates these possibilities. Given these possibilities, the code for the `removeMin` operation is relatively straightforward.

```
   /**
    * Removes the node with the least value from the binary search
    * tree and returns a reference to its element. Throws an
    * EmptyBinarySearchTreeException if this tree is empty.
    *
    * @return                          a reference to the node with the least
    *                                  value
    * @throws EmptyCollectionException if an empty collection exception occurs
    */
```

```
public T removeMin() throws EmptyCollectionException
{
   T result = null;

   if (isEmpty())
      throw new EmptyCollectionException ("binary search tree");
   else
   {
      if (root.left == null)
      {
         result = root.element;
         root = root.right;
      }
      else
      {
         BinaryTreeNode<T> parent = root;
         BinaryTreeNode<T> current = root.left;
         while (current.left != null)
         {
            parent = current;
            current = current.left;
         }
         result = current.element;
         parent.left = current.right;
      }

      count--;
   }

   return result;
}
```

The `removeMax`, `findMin`, and `findMax` operations are left as exercises.

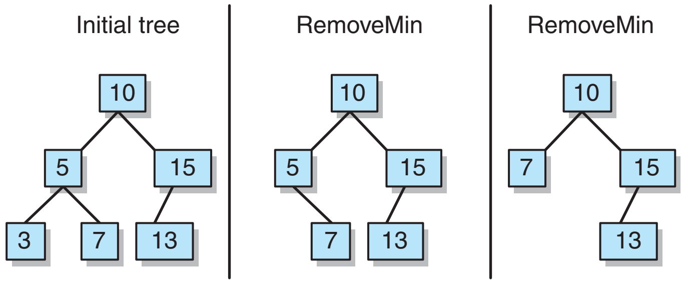


FIGURE 10.5 Removing the minimum element from a binary search tree

10.3 Implementing Binary Search Trees: With Arrays

In Chapter 9, we introduced a simple implementation of an `ArrayBinaryTree` class. Each element was stored in the tree using a computational strategy to maintain the relationship between parents and children in the tree (e.g., left child at position 2n + 1 and right child at position 2(n + 1)). We can simply extend that definition with an `ArrayBinarySearchTree` class implementing the `BinarySearchTreeADT` interface. Because we are extending the `ArrayBinaryTree` class from Chapter 9, all of the methods we discussed are still supported, including the various traversals.

Our `ArrayBinarySearchTree` class offers two constructors: one to create an empty `ArrayBinarySearchTree` and the other to create an `ArrayBinarySearchTree` with a particular element at the root. Both of these constructors simply refer to the equivalent constructors of the super class (i.e., the `ArrayBinaryTree` class). Note that we are also keeping track of the height of the tree and the maximum index used in the array.

```
/**
 * Creates an empty binary search tree.
 */
public ArrayBinarySearchTree()
{
   super();
   height = 0;
   maxIndex = -1;
}

/**
 * Creates a binary search with the specified element as its
 * root.
 *
 * @param element the element that will become the root of the new tree
 */
public ArrayBinarySearchTree (T element)
{
   super(element);
   height = 1;
   maxIndex = 0;
}
```

The addElement Operation

The `addElement` method adds a given element to an appropriate location in the tree, given its value. If the element is not `Comparable`, the method throws a `ClassCastException`. If the tree is empty, the new element becomes the root. If the tree is not empty, the new element is compared to the element at the root. If it is less than the element stored at the root and the left child of the root is null, then the new element becomes the left child of the root. If the new element is less than the element stored at the root and the left child of the root is not null, then we traverse to the left child of the root and compare again. If the new element is greater than or equal to the element stored at the root and the right child of the root is null, then the new element becomes the right child of the root. If the new element is greater than or equal to the element stored at the root and the right child of the root is not null, then we traverse to the right child of the root and compare again. Again, we have chosen to implement this method iteratively. Creating the recursive implementation is left as a programming project.

```
   /**
    * Adds the specified object to this binary search tree in the
    * appropriate position according to its key value.  Note that
    * equal elements are added to the right.  Also note that the
    * index of the left child of the current index can be found by
    * doubling the current index and adding 1.  Finding the index
    * of the right child can be calculated by doubling the current
    * index and adding 2.
    *
    * @param element the element to be added to the search tree
    */
   public void addElement (T element)
   {
      if (tree.length < maxIndex*2+3)
         expandCapacity();

      Comparable<T> tempElement = (Comparable<T>)element;

      if (isEmpty())
      {
         tree[0] = element;
         maxIndex = 0;
      }
      else
      {
         boolean added = false;
         int currentIndex = 0;
         while (!added)
         {
            if (tempElement.compareTo((tree[currentIndex]) ) < 0)
            {

               // go left
               if (tree[currentIndex*2+1] == null)
               {
                  tree[currentIndex*2+1] = element;
                  added = true;
                  if (currentIndex*2+1 > maxIndex)
                     maxIndex = currentIndex*2+1;
               }
               else
                  currentIndex = currentIndex*2+1;
            }

            else
            {

               // go right
               if (tree[currentIndex*2+2] == null)
               {
                  tree[currentIndex*2+2] = element;
                  added = true;
                  if (currentIndex*2+2 > maxIndex)
                     maxIndex = currentIndex*2+2;
               }
               else
                  currentIndex = currentIndex*2+2;
            }
         }
      }
      height = (int)(Math.log(maxIndex + 1) / Math.log(2)) + 1;
      count++;
   }
```

The `removeElement` Operation

The `removeElement` method removes a given `Comparable` element from a binary search tree or throws an `ElementNotFoundException` if the given target is not found in the tree. Like we did with the linked implementation, we must reorder the tree to maintain its structure after the removal. As we did with the linked implementation, we will use a protected method to replace the element that is being removed. We will also use a protected method that will find the array index of the given element if it is in the tree.

```
/**
 * Removes the first element that matches the specified target
 * element from this binary search tree and returns a reference to
 * it.  Throws an ElementNotFoundException if the specified target
 * element is not found in the binary search tree.
 *
 * @param targetElement              the element to be removed from the tree
 * @return                           a reference to the removed element
 * @throws ElementNotFoundException  if an element not found exception occurs
 */
public T removeElement  (T targetElement)
                        throws ElementNotFoundException
{
   T result = null;
   boolean found = false;

   if (isEmpty())
      throw new ElementNotFoundException("binary search tree");

   Comparable<T> tempElement = (Comparable<T>)targetElement;

   int targetIndex = findIndex (tempElement, 0);

   result = tree[targetIndex] ;
   replace(targetIndex);
   count--;

   int temp = maxIndex;
   maxIndex = -1;
   for (int i = 0; i <= temp; i++)
   {
      if (tree[i] != null)
         maxIndex = i;
   }

   height = (int)(Math.log(maxIndex + 1) / Math.log(2)) + 1;

   return result;
}
```

The following code illustrates the method to find the index of a given element if it is in the tree.

```
/**
 * Returns the index of the specified target element if it is
 * found in this binary tree.  Throws an ElementNotFoundException if
 * the specified target element is not found in the binary tree.
 *
 * @param targetElement              the element being sought in the tree
 * @return                           true if the element is in the tree
 * @throws ElementNotFoundException  if an element not found exception occurs
 */
protected int findIndex (Comparable<T>  targetElement, int n) throws
ElementNotFoundException
{
   int result = 0;

   if (n > tree.length)
      throw new ElementNotFoundException("binary search tree");
   if (tree[n] == null)
      throw new ElementNotFoundException("binary search tree");
   if (targetElement.compareTo(tree[n]) == 0)
      result = n;
   else
      if (targetElement.compareTo(tree[n]) > 0)
         result = findIndex (targetElement, (2 * (n + 1)));
      else
         result = findIndex (targetElement, (2 * n + 1));

   return result;
}
```

The method to replace the element being removed is far more interesting in the array implementation than it was in the linked implementation. Keep in mind, in this implementation it is not just a matter of finding the replacement element and moving it, a significant portion of the array, from the point of the deletion forward, will have to be reordered to maintain our computational strategy. We have chosen to use a series of unordered lists to support this operation.

```
/**
 * Removes the node specified for removal and shifts the tree
 * array accordingly.
 *
 * @param targetIndex the node to be removed
 */
```

```
protected void replace (int targetIndex)
{
   int currentIndex, parentIndex, temp, oldIndex, newIndex;
   ArrayUnorderedList<Integer> oldlist = new ArrayUnorderedList<Integer>();
   ArrayUnorderedList<Integer> newlist = new ArrayUnorderedList<Integer>();
   ArrayUnorderedList<Integer> templist = new ArrayUnorderedList<Integer>();
   Iterator<Integer> oldIt, newIt;

   // if target node has no children

   if ((targetIndex*2+1 >= tree.length) || (targetIndex*2+2 >= tree.length))
      tree[targetIndex] = null;

   // if target node has no children
   else if ((tree[targetIndex*2+1] == null) && (tree[targetIndex*2+2] == null))
      tree[targetIndex] = null;

   // if target node only has a left child
   else if ((tree[targetIndex*2+1] != null) && (tree[targetIndex*2+2] == null))
   {

      // fill newlist with indices of nodes that will replace
      // the corresponding indices in oldlist
      currentIndex = targetIndex*2+1;
      templist.addToRear(new Integer(currentIndex));
      while (!templist.isEmpty())
      {
         currentIndex = ((Integer)templist.removeFirst()).intValue();
         newlist.addToRear(new Integer(currentIndex));
         if ((currentIndex*2+2) <= (Math.pow(2,height)-2))
         {
            templist.addToRear(new Integer(currentIndex*2+1));
            templist.addToRear(new Integer(currentIndex*2+2));
         }
      }

      // fill oldlist
      currentIndex = targetIndex;
      templist.addToRear(new Integer(currentIndex));
      while (!templist.isEmpty())
      {
         currentIndex = ((Integer)templist.removeFirst()).intValue();
         oldlist.addToRear(new Integer(currentIndex));
         if ((currentIndex*2+2) <= (Math.pow(2,height)-2))
```

```
            {
               templist.addToRear(new Integer(currentIndex*2+1));
               templist.addToRear(new Integer(currentIndex*2+2));
            }
         }

         // do replacement
         oldIt = oldlist.iterator();
         newIt = newlist.iterator();
         while (newIt.hasNext())
         {
            oldIndex = oldIt.next();
            newIndex = newIt.next();
            tree[oldIndex] = tree[newIndex];
            tree[newIndex] = null;
         }
      }

      // if target node only has a right child
      else if ((tree[targetIndex*2+1] == null) && (tree[targetIndex*2+2] != null))
      {

         // fill newlist with indices of nodes that will replace
         // the corresponding indices in oldlist
         currentIndex = targetIndex*2+2;
         templist.addToRear(new Integer(currentIndex));
         while (!templist.isEmpty())
         {
            currentIndex = ((Integer)templist.removeFirst()).intValue();
            newlist.addToRear(new Integer(currentIndex));
            if ((currentIndex*2+2) <= (Math.pow(2,height)-2))
            {
               templist.addToRear(new Integer(currentIndex*2+1));
               templist.addToRear(new Integer(currentIndex*2+2));
            }
         }

         // fill oldlist
         currentIndex = targetIndex;
         templist.addToRear(new Integer(currentIndex));
         while (!templist.isEmpty())
         {
            currentIndex = ((Integer)templist.removeFirst()).intValue();
            oldlist.addToRear(new Integer(currentIndex));
            if ((currentIndex*2+2) <= (Math.pow(2,height)-2))
      {
         templist.addToRear(new Integer(currentIndex*2+1));
         templist.addToRear(new Integer(currentIndex*2+2));
      }
   }

   // do replacement
   oldIt = oldlist.iterator();
   newIt = newlist.iterator();
   while (newIt.hasNext())
   {
      oldIndex = oldIt.next();

      newIndex = newIt.next();
      tree[oldIndex] = tree[newIndex];
      tree[newIndex] = null;
   }
}

// if target node has two children
else
{
   currentIndex = targetIndex*2+2;

   while (tree[currentIndex*2+1] != null)
      currentIndex = currentIndex*2+1;

   tree[targetIndex] = tree[currentIndex];

   // the index of the root of the subtree to be replaced
   int currentRoot = currentIndex;

   // if currentIndex has a right child
   if (tree[currentRoot*2+2] != null)
   {

      // fill newlist with indices of nodes that will replace
      // the corresponding indices in oldlist
      currentIndex = currentRoot*2+2;
      templist.addToRear(new Integer(currentIndex));
      while (!templist.isEmpty())
      {
         currentIndex = ((Integer)templist.removeFirst()).intValue();
         newlist.addToRear(new Integer(currentIndex));
         if ((currentIndex*2+2) <= (Math.pow(2,height)-2))
```

```
               {
                  templist.addToRear(new Integer(currentIndex*2+1));
                  templist.addToRear(new Integer(currentIndex*2+2));
               }
            }

            // fill oldlist
            currentIndex = currentRoot;
            templist.addToRear(new Integer(currentIndex));
            while (!templist.isEmpty())
            {
               currentIndex = ((Integer)templist.removeFirst()).intValue();
               oldlist.addToRear(new Integer(currentIndex));
               if ((currentIndex*2+2) <= (Math.pow(2,height)-2))
               {
                  templist.addToRear(new Integer(currentIndex*2+1));
                  templist.addToRear(new Integer(currentIndex*2+2));
               }
            }

            // do replacement
            oldIt = oldlist.iterator();
            newIt = newlist.iterator();
            while (newIt.hasNext())
            {
               oldIndex = oldIt.next();
               newIndex = newIt.next();


               tree[oldIndex] = tree[newIndex];
               tree[newIndex] = null;
            }
         }
         else
            tree[currentRoot] = null;
      }
   }
```

The `removeAllOccurrences` Operation

The `removeAllOccurrences` method removes all occurrences of a given element from a binary search tree and throws an `ElementNotFoundException` if the given element is not found in the tree. This method also throws a `ClassCastException` if the element given is not `Comparable`. This method makes use of the `removeElement` method by calling it once, guaranteeing that the exception will be thrown if there is not at least one occurrence of the element in the tree. The `removeElement` method is then called again as long as the tree contains the target element. Note that the code for this method is identical to that of the linked implementation.

```
/**
 * Removes elements that match the specified target element from
 * the binary search tree. Throws a ElementNotFoundException if
 * the specified target element is not found in this tree.
 *
 * @param targetElement             the element being sought in the binary
 *                                  search tree
 * @throws ElementNotFoundException  if an element not found exception occurs
 */
public void removeAllOccurrences  (T targetElement)
                                  throws ElementNotFoundException
{
   removeElement(targetElement);

   try
   {
      while (contains( (T) targetElement))
         removeElement(targetElement);
   }

   catch (Exception ElementNotFoundException)
   {
   }
}
```

The `removeMin` Operation

There are three possible cases for the location of the minimum element in a binary search tree:

- If the root has no left child, then the root is the minimum element and the right child of the root becomes the new root.
- If the leftmost node of the tree is a leaf, then it is the minimum element, and we simply set its parent's left child reference to null.
- If the leftmost node of the tree is an internal node, then we set its parent's left child reference to point to the right child of the node to be removed.

Like the `removeElement` method, the `removeMin` method makes use of the `protected replace` method to replace the element being removed from the tree.

```
/**
 * Removes the node with the least value from this binary search
 * tree and returns a reference to its element. Throws an
 * EmptyBinarySearchTreeException if the binary search tree is
 * empty.
 *
 * @return a reference to the node with the least value in this tree
 * @throws EmptyCollectionException if an empty collection exception
occurs
 */
public T removeMin() throws EmptyCollectionException
{
   T result = null;

   if (isEmpty())
      throw new EmptyCollectionException ("binary search tree");
   else
   {
      int currentIndex = 1;
      int previousIndex = 0;
      while (tree[currentIndex] != null && currentIndex <= tree.length)

      {
         previousIndex = currentIndex;
         currentIndex = currentIndex * 2 + 1;
      }
      result = tree[previousIndex] ;
      replace(previousIndex);
   }

   count--;

   return result;
}
```

The `removeMax`, `findMin`, and `findMax` operations are left as exercises.

10.4 Using Binary Search Trees: Implementing Ordered Lists

As we discussed in Chapter 9, one of the uses of trees is to provide efficient implementations of other collections. The `OrderedList` collection from Chapter 6 provides an excellent example. Figure 10.6 reminds us of the common operations for lists, and Figure 10.7 reminds us of the operations particular to an ordered list.

Operation	Description
`removeFirst`	Removes the first element from the list.
`removeLast`	Removes the last element from the list.
`remove`	Removes a particular element from the list.
`first`	Examines the element at the front of the list.
`last`	Examines the element at the rear of the list.
`contains`	Determines if the list contains a particular element.
`isEmpty`	Determines if the list is empty.
`size`	Determines the number of elements on the list.

FIGURE 10.6 The common operations on a list

Using a binary search tree, we can create an implementation called `BinarySearchTreeList` that is a more efficient implementation than those we discussed in Chapter 6.

For simplicity, we have implemented both the `ListADT` and the `OrderedListADT` interfaces with the `BinarySearchTreeList` class, as shown in Listing 10.2. For some of the methods, the same method from either the `LinkedBinaryTree` or `LinkedBinarySearchTree` class will suffice. This is the case for the `contains`, `isEmpty`, and `size` operations. For the rest of the operations, there is a one-to-one correspondence between methods of the `LinkedBinaryTree` or `LinkedBinarySearchTree` classes and the required methods for an ordered list. Thus, each of these methods is implemented by simply calling the associated method for a `LinkedBinarySearchTree`. This is the case for the `add`, `removeFirst`, `removeLast`, `remove`, `first`, `last`, and `iterator` methods.

KEY CONCEPT
One of the uses of trees is to provide efficient implementations of other collections.

LISTING 10.2

```
/**
 * BinarySearchTreeList represents an ordered list implemented using a binary
 * search tree.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 8/19/08
 */

package jss2;
import jss2.exceptions.*;
import java.util.Iterator;
```

LISTING 10.2 *continued*

```
public class BinarySearchTreeList<T> extends LinkedBinarySearchTree<T>
                  implements ListADT<T>, OrderedListADT<T>, Iterable<T>
{

   /**
    * Creates an empty BinarySearchTreeList.
    */
   public BinarySearchTreeList()
   {
      super();
   }

   /**
    * Adds the given element to this list.
    *
    * @param element the element to be added to this list
    */
   public void add (T element)
   {
      addElement(element);
   }

   /**
    * Removes and returns the first element from this list.
    *
    * @return the first element in this list
    */
   public T removeFirst ()
   {
      return removeMin();
   }

   /**
    * Removes and returns the last element from this list.
    *
    * @return the last element from this list
    */
   public T removeLast ()
   {
      return removeMax();
   }
```

LISTING 10.2 *continued*

```
   /**
    * Removes and returns the specified element from this list.
    *
    * @param element  the element being sought in this list
    * @return         the element from the list that matches the target
    */
   public T remove (T element)
   {
      return removeElement(element);
   }

   /**
    * Returns a reference to the first element on this list.
    *
    * @return a reference to the first element in this list
    */
   public T first ()
   {
      return findMin();
   }

   /**
    * Returns a reference to the last element on this list.
    *
    * @return a reference to the last element in this list
    */
   public T last ()
   {
      return findMax();
   }

   /**
    * Returns an iterator for the list.
    *
    * @return an iterator over the elements in this list
    */
   public Iterator<T> iterator()
   {
      return iteratorInOrder();
   }
}
```

Operation	Description
`add`	Adds an element to the list.

FIGURE 10.7 The operation particular to an ordered list

Analysis of the `BinarySearchTreeList` Implementation

We will assume that the `LinkedBinarySearchTree` implementation used in the `BinarySearchTreeList` implementation is a balanced binary search tree with the added property that the maximum depth of any node is $\log_2(n)$, where n is the number of elements stored in the tree. This is a tremendously important assumption, as we will see over the next several sections. With that assumption, Figure 10.8 shows a comparison of the order of each operation for a singly linked implementation of an ordered list and our `BinarySearchTreeList` implementation.

Note that given our assumption of a balanced binary search tree, both the `add` and `remove` operations could cause the tree to need to be rebalanced, which, depending on the algorithm used, could affect the analysis. It is also important to note that although some operations are more efficient in the tree implementation, such as `removeLast`, `last`, and `contains`, others, such as `removeFirst` and `first`, are less efficient when implemented using a tree.

Operation	LinkedList	BinarySearchTreeList
`removeFirst`	O(1)	O(log n)
`removeLast`	O(n)	O(log n)
`remove`	O(n)	O(log n)*
`first`	O(1)	O(log n)
`last`	O(n)	O(log n)
`contains`	O(n)	O(log n)
`isEmpty`	O(1)	O(1)
`size`	O(1)	O(1)
`add`	O(n)	O(log n)*
*both the `add` and `remove` operations may cause the tree to become unbalanced		

FIGURE 10.8 Analysis of linked list and binary search tree implementations of an ordered list

10.5 Balanced Binary Search Trees

Why is our balance assumption important? What would happen to our analysis if the tree were not balanced? As an example, let's assume that we read the following list of integers from a file and added them to a binary search tree:

3 5 9 12 18 20

Figure 10.9 shows the resulting binary search tree. This resulting binary tree, referred to as a *degenerate tree*, looks more like a linked list, and in fact is less efficient than a linked list because of the additional overhead associated with each node.

If this is the tree we are manipulating, then our analysis from the previous section will look far worse. For example, without our balance assumption, the `addElement` operation would have worst case time complexity of O(n) instead of O(log n) because of the possibility that the root is the smallest element in the tree and the element we are inserting might be the largest element.

KEY CONCEPT

If a binary search tree is not balanced, it may be less efficient than a linear structure.

Our goal instead is to keep the maximum path length in the tree at or near $\log_2 n$. There are a variety of algorithms available for balancing or maintaining balance in a tree. There are brute-force methods, which are not elegant or efficient, but get the job done. For example, we could write an inorder traversal of the tree to an array and then use a recursive method (much like binary search) to insert the middle element of the array as the root, and then build balanced left and right subtrees. Though such an approach would work, there are more elegant solutions, such as AVL trees and red/black trees, which we examine later in this chapter.

However, before we move on to these techniques, we need to understand some additional terminology that is common to many balancing techniques. The methods

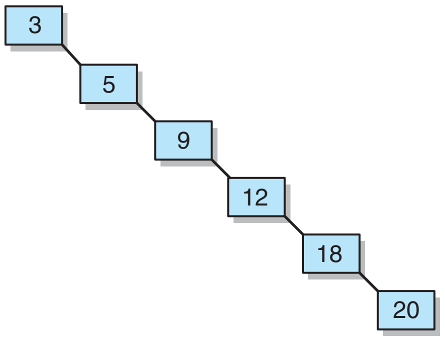


FIGURE 10.9 A degenerate binary tree

described here will work for any subtree of a binary search tree as well. For those subtrees, we simply replace the reference to root with the reference to the root of the subtree.

Right Rotation

Figure 10.10 shows a binary search tree that is not balanced and the processing steps necessary to rebalance it. The maximum path length in this tree is 3 and the minimum path length is 1. With only 6 elements in the tree, the maximum path length should be $\log_2 6$ or 2. To get this tree into balance, we need to

- Make the left child element of the root the new root element.
- Make the former root element the right child element of the new root.
- Make the right child of what was the left child of the former root the new left child of the former root.

This is referred to as a *right rotation* and is often referred to as a right rotation of the left child around the parent. The last image in Figure 10.10 shows the same tree after a right rotation. The same kind of rotation can be done at any level of the tree. This single rotation to the right will solve the imbalance if the imbalance is caused by a long path length in the left subtree of the left child of the root.

Left Rotation

Figure 10.11 shows another binary search tree that is not balanced. Again, the maximum path length in this tree is 3 and the minimum path length is 1.

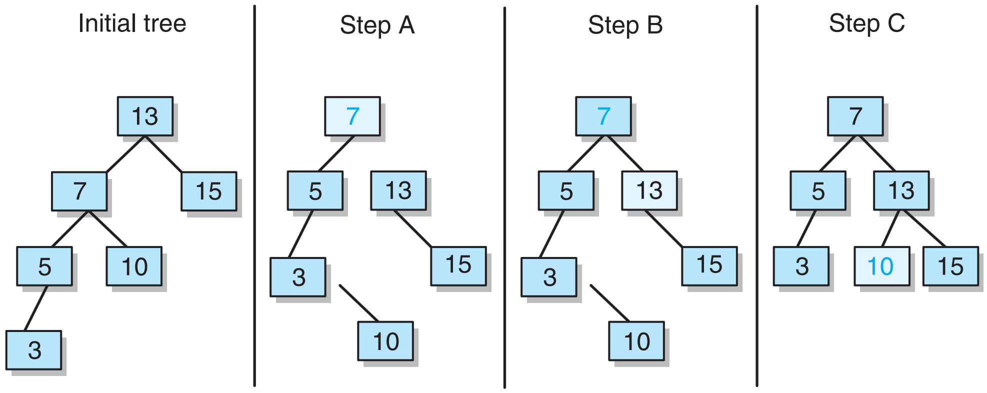


FIGURE 10.10 Unbalanced tree and balanced tree after a right rotation

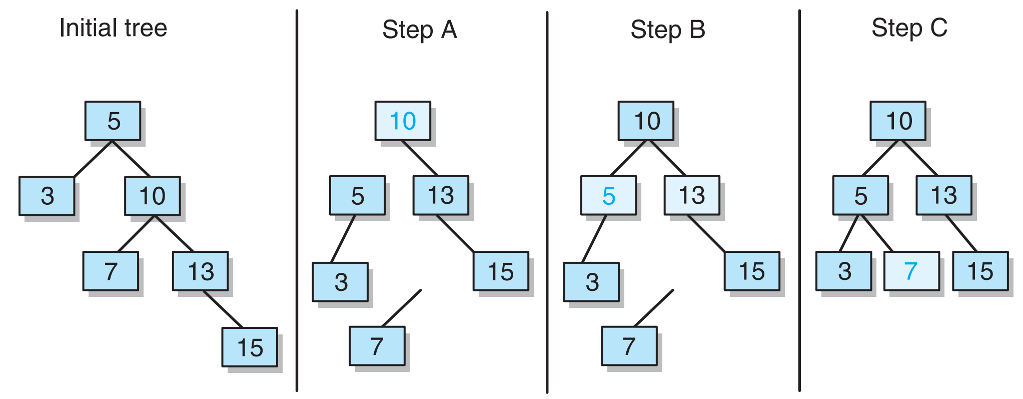


FIGURE 10.11 Unbalanced tree and balanced tree after a left rotation

However, this time the larger path length is in the right subtree of the right child of the root. To get this tree into balance, we need to

- Make the right child element of the root the new root element.
- Make the former root element the left child element of the new root.
- Make the left child of what was the right child of the former root the new right child of the former root.

This is referred to as a *left rotation* and is often stated as a left rotation of the right child around the parent. Figure 10.11 shows the same tree through the processing steps of a left rotation. The same kind of rotation can be done at any level of the tree. This single rotation to the left will solve the imbalance if the imbalance is caused by a longer path length in the right subtree of the right child of the root.

Rightleft Rotation

Unfortunately, not all imbalances can be solved by single rotations. If the imbalance is caused by a long path length in the left subtree of the right child of the root, we must first perform a right rotation of the left child of the right child of the root around the right child of the root, and then perform a left rotation of the resulting right child of the root around the root. Figure 10.12 illustrates this process.

Leftright Rotation

Similarly, if the imbalance is caused by a long path length in the right subtree of the left child of the root, we must first perform a left rotation of the right child of the left child of the root around the left child of the root, and then perform a right

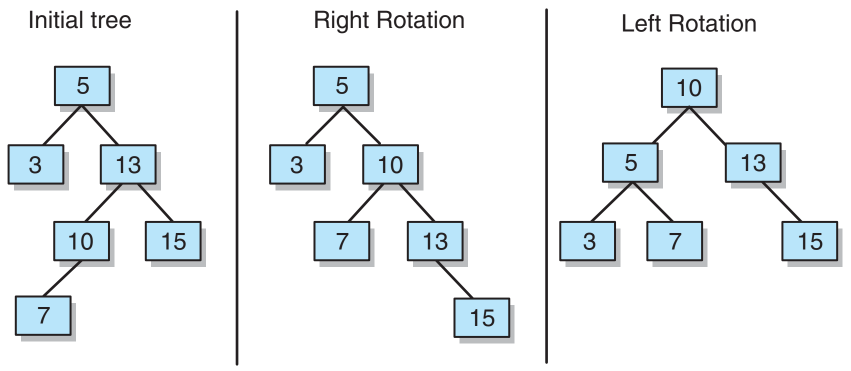


FIGURE 10.12 A rightleft rotation

rotation of the resulting left child of the root around the root. Figure 10.13 illustrates this process.

10.6 Implementing Binary Search Trees: AVL Trees

We have been discussing a generic method for balancing a tree where the maximum path length from the root must be no more than $\log_2 n$ and the minimum path length from the root must be no less than $\log_2 n-1$. Adel'son-Vel'skii and Landis developed a method called *AVL trees* that is a variation on this theme. For each node in the tree, we will keep track of the height of the left and right subtrees. For any node in the tree, if the *balance factor*, or the difference in the heights of its subtrees (height of the right subtree minus the height of the left subtree), is greater

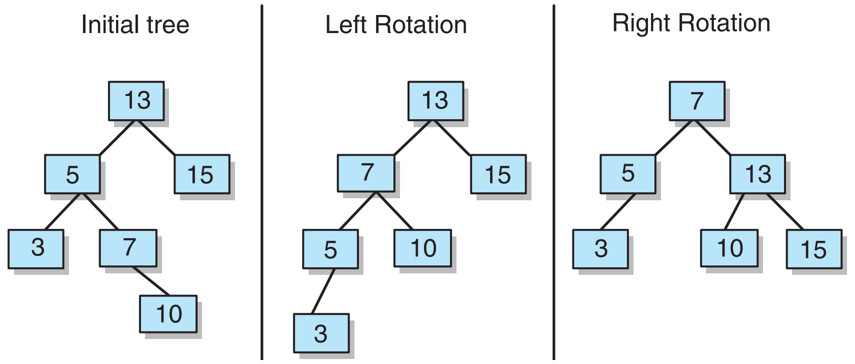


FIGURE 10.13 A leftright rotation

than 1 or less than –1, then the subtree with that node as the root needs to be rebalanced.

There are only two ways that a tree, or any subtree of a tree, can become unbalanced: through the insertion of a node or through the deletion of a node. Thus, each time one of these operations is performed, the balance factors must be updated and the balance of the tree must be checked starting at the point of insertion or removal of a node and working up toward the root of the tree. Because of this need to work back up the tree, AVL trees are often best implemented by including a parent reference in each node. In the diagrams that follow, all edges are represented as a single bidirectional line.

The cases for rotation that we discussed in the last section apply here as well, and by using this method, we can easily identify when to use each.

KEY CONCEPT
The height of the right subtree minus the height of the left subtree is called the balance factor of a node.

KEY CONCEPT
There are only two ways that a tree, or any subtree of a tree, can become unbalanced: through the insertion of a node or through the deletion of a node.

Right Rotation in an AVL Tree

If the balance factor of a node is –2, this means that the node's left subtree has a path that is too long. We then check the balance factor of the left child of the original node. If the balance factor of the left child is –1, this means that the long path is in the left subtree of the left child and therefore a simple right rotation of the left child around the original node will rebalance the tree. Figure 10.14 shows

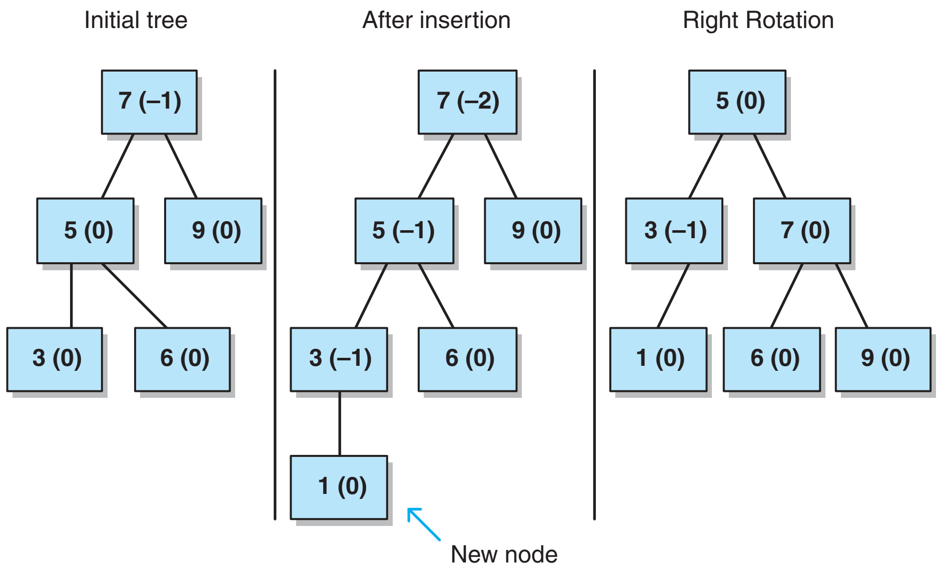


FIGURE 10.14 A right rotation in an AVL tree

how an insertion of a node could cause an imbalance and how a right rotation would resolve it. Note that we are representing both the values stored at each node and the balance factors, with the balance factors shown in parentheses.

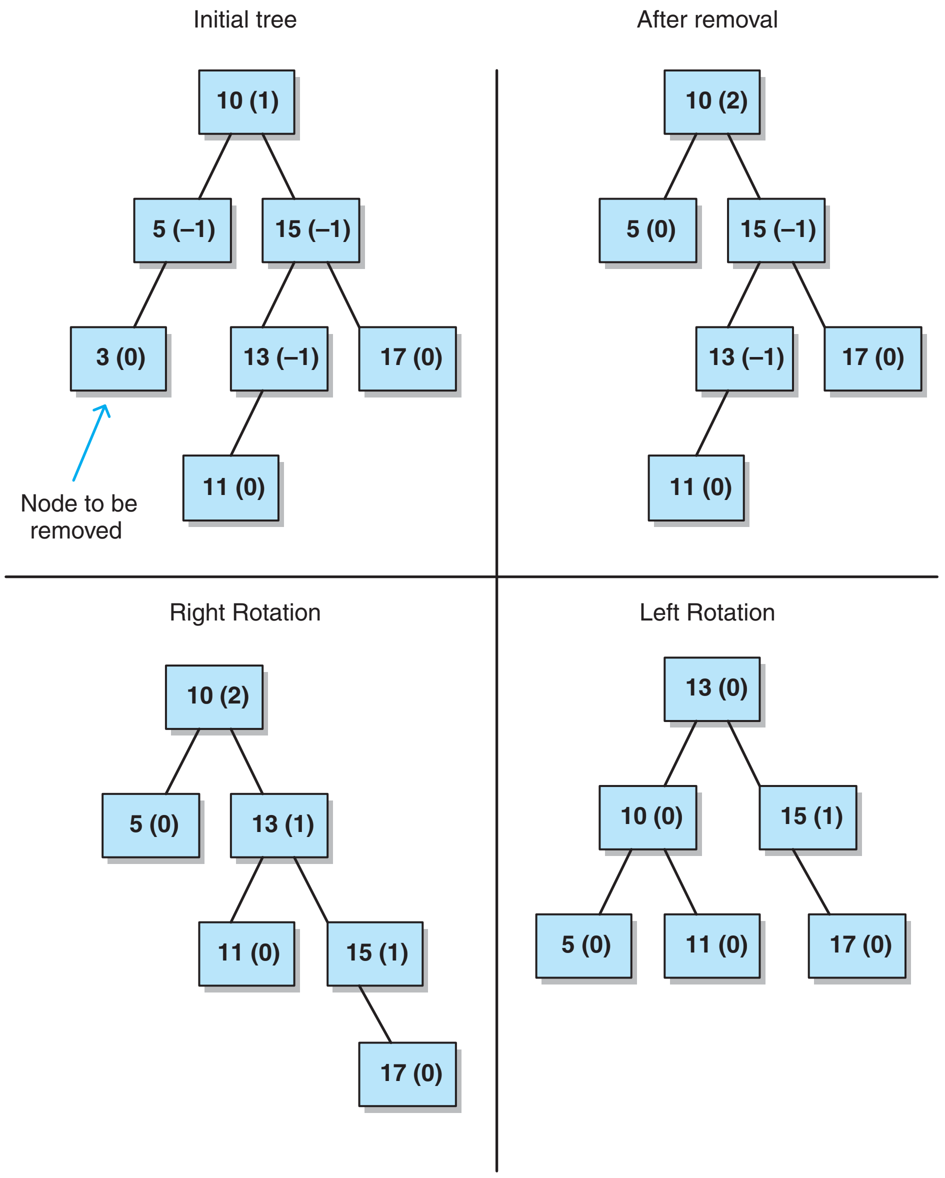


FIGURE 10.15 A rightleft rotation in an AVL tree

Left Rotation in an AVL Tree

If the balance factor of a node is +2, this means that the node's right subtree has a path that is too long. We then check the balance factor of the right child of the original node. If the balance factor of the right child is +1, this means that the long path is in the right subtree of the right child and therefore a simple left rotation of the right child around the original node will rebalance the tree.

Rightleft Rotation in an AVL Tree

If the balance factor of a node is +2, this means that the node's right subtree has a path that is too long. We then check the balance factor of the right child of the original node. If the balance factor of the right child is –1, this means that the long path is in the left subtree of the right child and therefore a rightleft double rotation will rebalance the tree. This is accomplished by first performing a right rotation of the left child of the right child of the original node around the right child of the original node, and then performing a left rotation of the right child of the original node around the original node. Figure 10.15 shows how the removal of an element from the tree could cause an imbalance and how a rightleft rotation would resolve it. Again, note that we are representing both the values stored at each node and the balance factors, with the balance factors shown in parentheses.

Leftright Rotation in an AVL Tree

If the balance factor of a node is –2, this means that the node's left subtree has a path that is too long. We then check the balance factor of the left child of the original node. If the balance factor of the left child is +1, this means that the long path is in the right subtree of the left child and therefore a leftright double rotation will rebalance the tree. This is accomplished by first performing a left rotation of the right child of the left child of the original node around the left child of the original node, and then performing a right rotation of the left child of the original node around the original node.

10.7 Implementing Binary Search Trees: Red/Black Trees

Another alternative to the implementation of binary search trees is the concept of a *red/black tree*, developed by Bayer and extended by Guibas and Sedgewick. A red/black tree is a balanced binary search tree in which we will store a color with

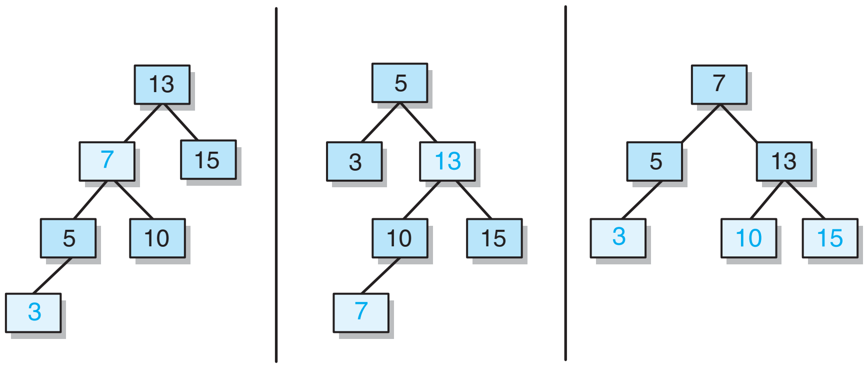


FIGURE 10.16 Valid red/black trees

each node (either red or black, usually implemented as a `boolean` value with false being equivalent to red). The following rules govern the color of a node:

- The root is black.
- All children of a red node are black.
- Every path from the root to a leaf contains the same number of black nodes.

Figure 10.16 shows three valid red/black trees (the lighter-shade nodes are "red"). Notice that the balance restriction on a red/black tree is somewhat less strict than that for AVL trees or for our earlier theoretical discussion. However, finding an element in both implementations is still an O(log n) operation. Because no red node can have a red child, then, at most, half of the nodes in a path could be red nodes and at least half of the nodes in a path are black. From this we can argue that the maximum height of a red/black tree is roughly 2*log n and thus the traversal of the longest path is still order log n.

KEY CONCEPT
The balance restriction on a red/black tree is somewhat less strict than that for AVL trees.

As with AVL trees, the only time we need to be concerned about balance is after an insertion or removal of an element in the tree. Unlike AVL trees, the two are handled quite separately.

Insertion into a Red/Black Tree

Insertion into a red/black tree will progress much as it did in our earlier `addElement` method. However, we will always begin by setting the color of the new element to red. Once the new element has been inserted, we will then rebalance the tree as needed and change the color of elements as needed to maintain the properties of a red/black tree. As a last step, we will always set the color of the root of the tree to black. For purposes of our discussion, we will simply refer to the color of a node as `node.color`. However, it may be more elegant in an actual implementation to create a method to return the color of a node.

The rebalancing (and recoloring) process after insertion is an iterative (or recursive) one starting at the point of insertion and working up the tree toward the root. Therefore, like AVL trees, red/black trees are best implemented by including a parent reference in each node. The termination conditions for this process are (`current == root`), where `current` is the node we are currently processing, or (`current.parent.color == black`) (i.e., the color of the parent of the current node is black). The first condition terminates the process because we will always set the root color to black, and the root is included in all paths and therefore cannot violate the rule that each path have the same number of black elements. The second condition terminates the process because the node pointed to by `current` will always be a red node. This means that if the parent of the current node is black, then all of the rules are met as well since a red node does not affect the number of black nodes in a path and because we are working from the point of insertion up, we will have already balanced the subtree under the current node.

In each iteration of the rebalancing process, we will focus on the color of the sibling of the parent of the current node. Keep in mind that there are two possibilities for the parent of the current node: `current.parent` could be a left child or a right child. Assuming that the parent of `current` is a right child, we can get the color information by using `current.parent.parent.left.color`, but for purposes of our discussion, we will use the terms `parentsleftsibling.color` and `parentsrightsibling.color`. It is also important to keep in mind that the color of a null element is considered to be black.

In the case where the parent of `current` is a right child, there are two cases, either (`parentsleftsibling.color == red`) or (`parentsleftsibling.color == black`). Keep in mind that in either case, we are describing processing steps that are occurring inside of a loop with the termination conditions described earlier. Figure 10.17 shows a red/black tree after insertion with this first case (`parentsleftsibling.color==red`). The processing steps in this case are

- Set the color of `current`'s parent to black.
- Set the color of `parentsleftsibling` to black.
- Set the color of `current`'s grandparent to red.
- Set `current` to point to the grandparent of `current`.

In Figure 10.17, we inserted 8 into our tree. Keep in mind that `current` points to our new node and `current.color` is set to red. Following the processing steps, we set the parent of `current` to black, we set the left sibling of the parent of `current` to black, and we set the grandparent of `current` to red. We then set

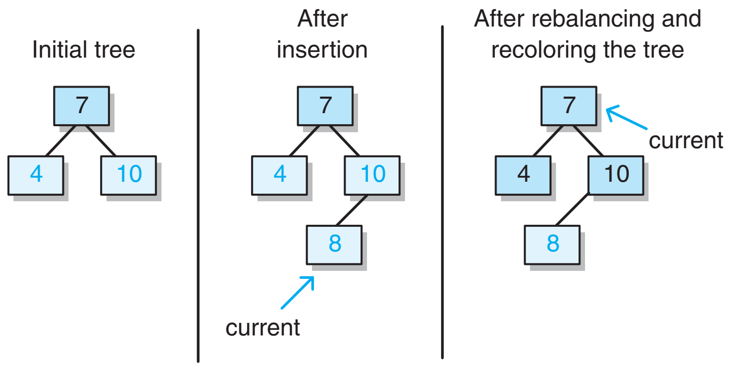


FIGURE 10.17 Red/black tree after insertion

`current` to point to the grandparent. Because the grandparent is the root, the loop terminates. Finally, we set the root of the tree to black.

However, if (`parentsleftsibling.color == black`), then we first need to check to see if `current` is a left or right child. If `current` is a left child, then we must set `current` equal to its parent and then rotate `current.left` to the right (around `current`) before continuing. Once this is accomplished, the processing steps are the same as if `current` were a right child to begin with:

- Set the color of `current`'s parent to black.
- Set the color of `current`'s grandparent to red.
- If `current`'s grandparent does not equal null, then rotate `current`'s parent to the left around `current`'s grandparent.

In the case where the parent of `current` is a left child, there are two cases, either (`parentsrightsibling.color == red`) or (`parentsrightsibling.color == black`). Keep in mind that in either case, we are describing processing steps that are occurring inside of a loop with the termination conditions described earlier. Figure 10.18 shows a red/black tree after insertion in this case (`parentsrightsibling.color==red`). The processing steps in this case are

- Set the color of `current`'s parent to black.
- Set the color of `parentsrightsibling` to black.
- Set the color of `current`'s grandparent to red.
- Set `current` to point to the grandparent of `current`.

In Figure 10.18. we inserted 5 into our tree, setting `current` to point to the new node and setting `current.color` to red. Again, following our processing steps, we set the parent of `current` to black, we set the right sibling of the parent

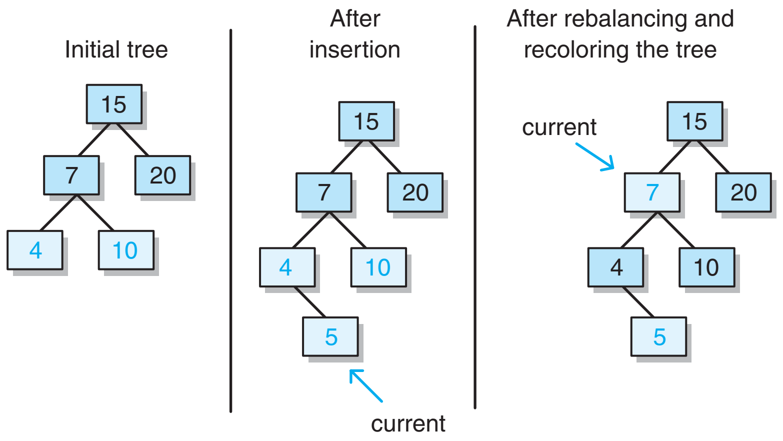


FIGURE 10.18 Red/black tree after insertion

of `current` to black, and we set the grandparent of `current` to red. We then set `current` to point to its grandparent. Because the parent of the new `current` is black, our loop terminates. Lastly, we set the color of the root to black.

If (`parentsrightsibling.color == black`), then we first need to check to see if `current` is a left or right child. If `current` is a right child, then we must set `current` equal to `current.parent` and then rotate `current.right` to the left (around `current`) before continuing. Once this is accomplished, the processing steps are the same as if `current` were a left child to begin with:

- Set the color of `current`'s parent to black.
- Set the color of `current`'s grandparent to red.
- If `current`'s grandparent does not equal null, then rotate `current`'s parent to the right around `current`'s grandparent.

As you can see, the cases, depending on whether or not `current`'s parent is a left or right child, are symmetrical.

Element Removal from a Red/Black Tree

As with insertion, the `removeElement` operation behaves much as it did before, only with the additional step of rebalancing (and recoloring) the tree. This rebalancing (and recoloring) process after removal of an element is an iterative one starting at the point of removal and working up the tree toward the root. Therefore, as stated earlier, red/black trees are often best implemented by including a parent reference in each node. The termination conditions for this process are (`current == root`), where `current` is the node we are currently processing, or (`current.color == red`).

As with the cases for insertion, the cases for removal are symmetrical depending upon whether `current` is a left or right child. We only examine the case where `current` is a right child. The other cases are easily derived by simply substituting left for right and right for left in the following cases.

In insertion, we were most concerned with the color of the sibling of the parent of the current node. For removal, we will focus on the color of the sibling of `current`. We could reference this color using `current.parent.left.color`, but we will simply refer to it as `sibling.color`. We will also look at the color of the children of the sibling. It is important to note that the default for color is black. Therefore, if at any time we are attempting to get the color of a null object, the result will be black. Figure 10.19 shows a red/black tree after the removal of an element.

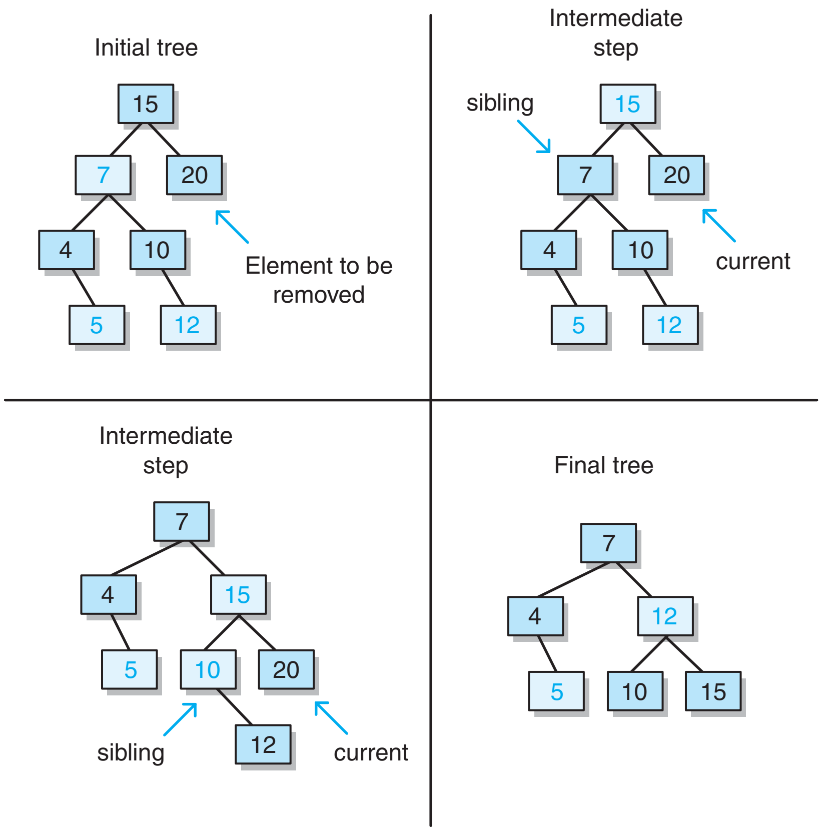


FIGURE 10.19 Red/black tree after removal

If the sibling's color is red, then before we do anything else, we must complete the following processing steps:

- Set the color of the sibling to black.
- Set the color of `current`'s parent to red.
- Rotate the sibling right around `current`'s parent.
- Set the sibling equal to the left child of `current`'s parent.

Next, our processing continues regardless of whether the original sibling was red or black. Now our processing is divided into one of two cases based upon the color of the children of the sibling. If both children of the sibling are black (or null), then we do the following:

- Set the color of the sibling to red.
- Set `current` equal to `current`'s parent.

If the children of the sibling are not both black, then we check to see if the left child of the sibling is black. If it is, we must complete the following steps before continuing:

- Set the color of the sibling's right child to black.
- Set the color of the sibling to red.
- Rotate the sibling's right child left around the sibling.
- Set the sibling equal to the left child of `current`'s parent.

Then to complete the process when both of the sibling's children are not black, we must:

- Set the color of the sibling to the color of `current`'s parent.
- Set the color of `current`'s parent to black.
- Set the color of the sibling's left child to black.
- Rotate the sibling right around `current`'s parent.
- Set `current` equal to the root.

Once the loop terminates, we must always then remove the node and set its parent's child reference to null.

10.8 Implementing Binary Search Trees: The Java Collections API

The Java Collections API provides two implementations of balanced binary search trees: `TreeSet` and `TreeMap`. Both use a red/black tree implementation approach. In order to understand these implementations, we must first discuss the difference between a *set* and a *map* in the Java Collections API.

KEY CONCEPT

The Java Collections API provides two implementations of balanced binary search trees, TreeSet and TreeMap, both of which use a red/black tree implementation.

In the terminology of the Java Collections API, all of the collections that we have discussed thus far could be considered sets (except that sets do not allow duplicates), because the data or element stored in each collection contains all of the data associated with that object. For example, if we were creating an ordered list of employees, ordered by name, then we would have created an `employee` object that contained all of the data for each employee, including the name and a `compareTo` method to test the name, and we would have used our operations for an ordered list to add those employees into the list.

DESIGN FOCUS

Java provides thorough and efficient implementations of binary search trees with the TreeSet and TreeMap classes. Why, then, do we spend time learning how to build such collections and learning other methods for balancing trees such as AVL trees? Languages come and go. Simply because Java is a popular language at the moment does not mean that it will continue to be or that developers will not be asked to use other languages, either newer or older languages, many of which do not provide tree implementations. Secondly, the principles involved in this discussion are perhaps more important than the implementations themselves. Understanding that a collection can be built to provide order, balance, and efficiency and the principles involved in how and why those things are done are important lessons.

However, in this same scenario, if we wanted to create an ordered list that is a map, we would have created a class to represent the name of each employee and a reference that would point to a second class that contains all of the rest of the employee data. We would have then used our ordered list operations to load the first class into our list, whereas the objects of the second class could exist anywhere in memory. The first class in this case is sometimes referred to as the *key*, whereas the second class is often referred to as the *data*.

In this way, as we manipulate elements of the list, we are only dealing with the key, the name, and the reference, which is a much smaller segment of memory than if we were manipulating all of the data associated with an employee. We also have the advantage that the same employee data could be referenced by multiple maps without having to make multiple copies. Thus, if for one application we wanted to represent employees in a stack collection and for another application we needed to represent employees as an ordered list, we could load keys into a stack and load matching keys into an ordered list while only having one instance of the actual data. Like any situation dealing with aliases (i.e., multiple references to the same object), we must be careful that changes to an object through one reference affect the object referenced by all of the other references because there is only one instance of the object.

Figures 10.20 and 10.21 show the operations for a `TreeSet` and `TreeMap`, respectively. Note that these implementations use (and allow the use of) a

Operation	**Description**
`TreeSet()`	Constructs a new, empty set, sorted according to the elements' natural order.
`TreeSet(Collection c)`	Constructs a new set containing the elements in the specified collection, sorted according to the elements' natural order.
`TreeSet(Comparator c)`	Constructs a new, empty set, sorted according to the given comparator.
`TreeSet(SortedSet s)`	Constructs a new set containing the same elements as the given sorted set, sorted according to the same ordering.
`boolean add(Object o)`	Adds the specified element to this set if it is not already present.
`boolean addAll(Collection c)`	Adds all of the elements in the specified collection to this set.
`void clear()`	Removes all of the elements from this set.
`Object clone()`	Returns a shallow copy of this `TreeSet` instance.
`Comparator comparator()`	Returns the comparator used to order this sorted set, or null if this `TreeSet` uses its elements' natural ordering.
`boolean contains(Object o)`	Returns true if this set contains the specified element.
`Object first()`	Returns the first (lowest) element currently in this sorted set.
`SortedSet headSet(Object toElement)`	Returns a view of the portion of this set whose elements are strictly less than `toElement`.
`boolean isEmpty()`	Returns true if this set contains no elements.
`Iterator iterator()`	Returns an iterator over the elements in this set.
`Object last()`	Returns the last (highest) element currently in this sorted set.
`boolean remove(Object o)`	Removes the given element from this set if it is present.
`int size()`	Returns the number of elements in this set (its cardinality).
`SortedSet subSet(Object fromElement, Object toElement)`	Returns a view of the portion of this set whose elements range from `fromElement`, inclusive, to `toElement`, exclusive.
`SortedSet tailSet(Object fromElement)`	Returns a view of the portion of this set whose elements are greater than or equal to `fromElement`.

FIGURE 10.20 Operations on a `TreeSet`

Operation	Description
`TreeMap()`	Constructs a new, empty map, sorted according to the keys' natural order.
`TreeMap(Comparator c)`	Constructs a new, empty map, sorted according to the given comparator.
`TreeMap(Map m)`	Constructs a new map containing the same mappings as the given map, sorted according to the keys' natural order.
`TreeMap(SortedMap m)`	Constructs a new map containing the same mappings as the given `SortedMap`, sorted according to the same ordering.
`void clear()`	Removes all mappings from this `TreeMap`.
`Object clone()`	Returns a shallow copy of this `TreeMap` instance.
`Comparator comparator()`	Returns the comparator used to order this map, or null if this map uses its keys' natural order.
`boolean containsKey(Object key)`	Returns true if this map contains a mapping for the specified key.
`boolean containsValue(Object value)`	Returns true if this map maps one or more keys to the specified value.
`Set entrySet()`	Returns a set view of the mappings contained in this map.
`Object firstKey()`	Returns the first (lowest) key currently in this sorted map.
`Object get(Object key)`	Returns the value to which this map maps the specified key.
`SortedMap headMap(Object toKey)`	Returns a view of the portion of this map whose keys are strictly less than `toKey`.
`Set keySet()`	Returns a set view of the keys contained in this map.
`Object lastKey()`	Returns the last (highest) key currently in this sorted map.
`Object put(Object key, Object value)`	Associates the specified value with the specified key in this map.
`void putAll(Map map)`	Copies all of the mappings from the specified map to this map.
`Object remove(Object key)`	Removes the mapping for this key from this `TreeMap` if present.
`int size()`	Returns the number of key-value mappings in this map.
`SortedMap subMap(Object fromKey, Object toKey)`	Returns a view of the portion of this map whose keys range from `fromKey`, inclusive, to `toKey`, exclusive.
`SortedMap tailMap(Object fromKey)`	Returns a view of the portion of this map whose keys are greater than or equal to `fromKey`.
`Collection values()`	Returns a collection view of the values contained in this map.

FIGURE 10.21 Operations on a `TreeMap`

Comparator instead of using Comparable as we did in our earlier implementations. The Comparator interface describes a method, compare, that, like compareTo, returns –1, 0, or 1, representing less than, equal to, or greater than. However, unlike compareTo, compare takes two arguments and does not need to be implemented within the class to be stored in the collection. We will discuss the general concepts of sets and maps further in Chapter 15 along with the Java Collections API implementations of these collections.

10.9 A Philosophical Quandary

Early in Chapter 3, we stated that our naming convention would follow that of the Java Collections API where a class implementing a collection would be given the name of the data structure used and the collection it implements (e.g., ArrayStack, LinkedList, ArrayQueue). Given that naming convention and what we have just learned about the Java Collections API implementations of trees (e.g., TreeSet and TreeMap), it would appear that we have a philosophical quandary. Is a tree a data structure or a collection? Earlier in this chapter, we presented our implementation of both an ArrayBinarySearchTree and a LinkedBinarySearchTree. This would suggest that a binary search tree is a collection. However, we also presented the BinarySearchTreeList class, which would suggest that a binary search tree is a data structure. Figure 10.22 illustrates how the BinarySearchTreeList class was built upon the LinkedBinary SearchTree class which was built upon the LinkedBinaryTree class and that class was built using references as links.

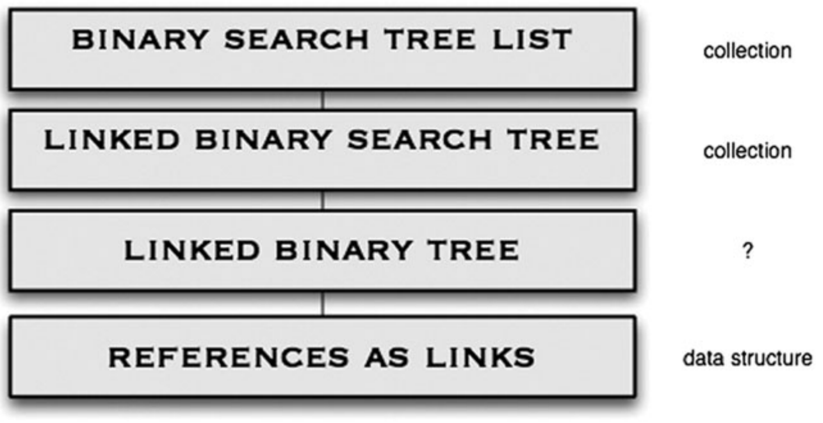


FIGURE 10.22 Using collections to implement other collections

So which of these represent data structures and which represents collections? Working from the top down, there is no question that an ordered list is a collection no matter how it is implemented. Similarly, even though it is often used to provide efficient implementations of other collections, a binary search tree fits our earlier definition of a collection by virtue of being an object that gathers and organizes other objects and by defining the specific ways in which those objects, which are called *elements* of the collection, can be accessed and managed. However, saying that a binary search tree is a collection does not mean that we cannot use it, or any other collection, to provide an implementation of other collections. It simply means that our naming convention can become unwieldy.

What about a binary tree? Keep in mind that without specifying any additional detail about the organization of a binary tree, we cannot create methods to add or remove elements from the tree. Our examples of using a binary tree thus far have been expression trees from Chapter 9 and binary search trees in this chapter. Although expression trees may come closer to a specific application, both expression trees and binary search trees could easily be considered collections. However, it is difficult to argue that a binary tree is in and of itself a collection, given that we cannot add or remove anything from it. Though it is certainly open for debate, perhaps the best categorization for a binary tree is as an *abstract data structure*.

Summary of Key Concepts

- A binary search tree is a binary tree with the added property that the left child is less than the parent, which is less than or equal to the right child.
- The definition of a binary search tree is an extension of the definition of a binary tree.
- Each `BinaryTreeNode` object maintains a reference to the element stored at that node as well as references to each of the node's children.
- In removing an element from a binary search tree, another node must be promoted to replace the node being removed.
- The leftmost node in a binary search tree will contain the minimum element, whereas the rightmost node will contain the maximum element.
- One of the uses of trees is to provide efficient implementations of other collections.
- If a binary search tree is not balanced, it may be less efficient than a linear structure.
- The height of the right subtree minus the height of the left subtree is called the balance factor of a node.
- There are only two ways that a tree, or any subtree of a tree, can become unbalanced: through the insertion of a node or through the deletion of a node.
- The balance restriction on a red/black tree is somewhat less strict than that for AVL trees. However, in both cases, the `find` operation is order log n.
- The Java Collections API provides two implementations of balanced binary search trees, `TreeSet` and `TreeMap`, both of which use a red/black tree implementation.

Self-Review Questions

SR 10.1 What is the difference between a binary tree and a binary search tree?

SR 10.2 Why are we able to specify `addElement` and `removeElement` operations for a binary search tree but we were unable to do so for a binary tree?

SR 10.3 Assuming that the tree is balanced, what is the time complexity (order) of the `addElement` operation?

SR 10.4 Without the balance assumption, what is the time complexity (order) of the `addElement` operation?

SR 10.5 As stated in this chapter, a degenerate tree might actually be less efficient than a linked list. Why?

SR 10.6 Our `removeElement` operation uses the inorder successor as the replacement for a node with two children. What would be another reasonable choice for the replacement?

SR 10.7 The `removeAllOccurrences` operation uses both the `contains` and `removeElement` operations. What is the resulting time complexity (order) for this operation?

SR 10.8 `RemoveFirst` and `first` were O(1) operations for our earlier implementation of an ordered list. Why are they less efficient for our `BinarySearchTreeOrderedList`?

SR 10.9 Why does the `BinarySearchTreeOrderedList` class have to define the `iterator` method? Why can't it just rely on the `iterator` method of its parent class like it does for `size` and `isEmpty`?

SR 10.10 What is the time complexity of the `addElement` operation after modifying to implement an AVL tree?

SR 10.11 What imbalance is fixed by a single right rotation?

SR 10.12 What imbalance is fixed by a leftright rotation?

SR 10.13 What is the balance factor of an AVL tree node?

SR 10.14 In our discussion of the process for rebalancing an AVL tree, we never discussed the possibility of the balance factor of a node being either +2 or –2 and the balance factor of one of its children being either +2 or –2. Why not?

SR 10.15 We noted that the balance restriction for a red/black tree is less strict than that of an AVL tree and yet we still claim that traversing the longest path in a red/black tree is still O(log n). Why?

SR 10.16 What is the difference between a `TreeSet` and a `TreeMap`?

Exercises

EX 10.1 Draw the binary search tree that results from adding the following integers (34 45 3 87 65 32 1 12 17). Assume our simple implementation with no balancing mechanism.

EX 10.2 Starting with the resulting tree from Exercise 10.1, draw the tree that results from removing (45 12 1), again using our simple implementation with no balancing mechanism.

EX 10.3 Repeat Exercise 10.1, this time assuming an AVL tree. Include the balance factors in your drawing.

EX 10.4 Repeat Exercise 10.2, this time assuming an AVL tree and using the result of Exercise 10.3 as a starting point. Include the balance factors in your drawing.

EX 10.5 Repeat Exercise 10.1, this time assuming a red/black tree. Label each node with its color.

EX 10.6 Repeat Exercise 10.2, this time assuming a red/black tree and using the result of Exercise 10.5 as a starting point. Label each node with its color.

EX 10.7 Starting with an empty red/black tree, draw the tree after insertion and before rebalancing, and after rebalancing (if necessary) for the following series of inserts and removals:

```
AddElement(40);
AddElement(25):
AddElement(10);
AddElement(5);
AddElement(1);
AddElement(45);
AddElement(50);
RemoveElement(40);
RemoveElement(25);
```

EX 10.8 Repeat Exercise 10.7, this time with an AVL tree.

Programming Projects

PP 10.1 The `ArrayBinarySearchTree` class is currently using the `find` and `contains` methods of the `ArrayBinaryTree` class. Implement these methods for the `ArrayBinarySearchTree` class so that they will be more efficient by making use of the ordering property of a binary search tree.

PP 10.2 The `LinkedBinarySearchTree` class is currently using the `find` and `contains` methods of the `LinkedBinaryTree` class. Implement these methods for the `LinkedBinarySearchTree` class so that they will be more efficient by making use of the ordering property of a binary search tree.

PP 10.3 Implement the `removeMax`, `findMin`, and `findMax` operations for our linked binary search tree implementation.

PP 10.4 Implement the `removeMax`, `findMin`, and `findMax` operations for our array binary search tree implementation.

PP 10.5 Implement a balance tree method for the linked implementation using the brute-force method described in Section 10.4.

PP 10.6 Implement a balance tree method for the array implementation using the brute-force method described in Section 10.4.

PP 10.7 Develop an array implementation of a binary search tree built upon an array implementation of a binary tree by using the simulated link strategy. Each element of the array will need to maintain both a reference to the data element stored there and the array positions of the left and right child. You also need to maintain a list of available array positions where elements have been removed, in order to reuse those positions.

PP 10.8 Modify the binary search tree implementation to make it an AVL tree.

PP 10.9 Modify the binary search tree implementation to make it a red/black tree.

PP 10.10 Modify the add operation for the linked implementation of a binary search tree to use a recursive algorithm.

PP 10.11 Modify the add operation for the array implementation of a binary search tree to use a recursive algorithm.

Answers to Self-Review Questions

SRA 10.1 A binary search tree has the added ordering property that the left child of any node is less than the node, and the node is less than or equal to its right child.

SRA 10.2 With the added ordering property of a binary search tree, we are now able to define what the state of the tree should be after an `add` or `remove`. We were unable to define that state for a binary tree.

SRA 10.3 If the tree is balanced, finding the insertion point for the new element will take at worst log n steps, and since inserting the element is simply a matter of setting the value of one reference, the operation is O(log n).

SRA 10.4 Without the balance assumption, the worst case would be a degenerate tree, which is effectively a linked list. Therefore, the `addElement` operation would be O(n).

SRA 10.5 A degenerate tree will waste space with unused references, and many of the algorithms will check for null references before following the degenerate path, thus adding steps that the linked list implementation does not have.

SRA 10.6 The best choice is the inorder successor since we are placing equal values to the right.

SRA 10.7 With our balance assumption, the `contains` operation uses the `find` operation, which will be rewritten in the `BinarySearchTree` class to take advantage of the ordering property and will be O(log n). The `removeElement` operation is O(log n). The `while` loop will iterate some constant (k) number of times depending on how many times the given element occurs within the tree. The worst case would be that all n elements of the tree are the element to be removed, which would make the tree degenerate, and in which case the complexity would be n*2*n or $O(n^2)$. However, the expected case would be some small constant (0<=k<n) occurrences of the element in a balanced tree, which would result in a complexity of k*2*log n or O(log n).

SRA 10.8 In our earlier linked implementation of an ordered list, we had a reference that kept track of the first element in the list, which made it quite simple to remove it or return it. With a binary search tree, we have to traverse to get to the leftmost element before knowing that we have the first element in the ordered list.

SRA 10.9 Remember that the iterators for a binary tree are all followed by which traversal order to use. That is why the `iterator` method for the `BinarySearchTreeOrderedList` class calls the `iteratorInOrder` method of the `BinaryTree` class.

SRA 10.10 Keep in mind that an `addElement` method only affects one path of the tree, which in a balanced AVL tree has a maximum length of log n. As we have discussed previously, finding the position to insert and setting the reference is O(log n). We then have to progress back up the same path, updating the balance factors of each node (if necessary) and rotating if necessary. Updating the balance factors is an O(1) step and rotation is also an O(1) step. Each of these will at most have to be done log n times. Therefore, `addElement` has time complexity 2*log n or O(log n).

SRA 10.11 A single right rotation will fix the imbalance if the long path is in the left subtree of the left child of the root.

SRA 10.12 A leftright rotation will fix the imbalance if the long path is in the right subtree of the left child of the root.

SRA 10.13 The balance factor of an AVL tree node is the height of the right subtree minus the height of the left subtree.

SRA 10.14 Rebalancing an AVL tree is done after either an insertion or a deletion and it is done starting at the affected node and working up along a single path to the root. As we progress upward, we update the balance factors and rotate if necessary. We will never encounter a situation where both a child and a parent have balance factors of +/–2 because we would have already fixed the child before we ever reached the parent.

SRA 10.15 Since no red node can have a red child, then at most half of the nodes in a path could be red nodes and at least half of the nodes in a path are black. From this we can argue that the maximum height of a red/black tree is roughly 2*log n and thus the traversal of the longest path is O(log n).

SRA 10.16 Both are red/black tree implementations of a binary search tree. The difference is that in a Set, all of the data are stored with an element, and with a `TreeMap`, a separate key is created and stored in the collection while the data are stored separately.

References

Adel'son-Vel'skii, G. M., and E. M. Landis. "An Algorithm for the Organization of Information." *Soviet Mathematics* 3 (1962): 1259–1263.

Bayer, R. "Symmetric Binary B-trees: Data Structure and Maintenance Algorithms." *Acta Informatica* (1972): 290–306.

Collins, W. J. *Data Structures and the Java Collections Framework*. New York: McGraw-Hill, 2002.

Cormen, T., C. Leierson, and R. Rivest. *Introduction to Algorithms*. New York: McGraw-Hill, 1992.

Guibas, L., and R. Sedgewick. "A Diochromatic Framework for Balanced Trees." *Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science* (1978): 8–21.

11 Priority Queues and Heaps

In this chapter, we will look at another ordered extension of binary trees. We will examine heaps, including both linked and array implementations, and the algorithms for adding and removing elements from a heap. We will also examine some uses for heaps including their principal use, priority queues.

CHAPTER OBJECTIVES

- Define a heap abstract data structure
- Demonstrate how a heap can be used to solve problems
- Examine various heap implementations
- Compare heap implementations

11.1 A Heap

A *heap* is a binary tree with two added properties:

- It is a complete tree, as described in Chapter 9.
- For each node, the node is less than or equal to both the left child and the right child.

KEY CONCEPT
A minheap is a complete binary tree in which each node is less than or equal to both of its children.

KEY CONCEPT
A minheap stores its smallest element at the root of the binary tree, and both children of the root of a minheap are also minheaps.

This definition describes a *minheap*. A heap can also be a *maxheap*, in which the node is greater than or equal to its children. We will focus our discussion in this chapter on minheaps. All of the same processes work for maxheaps by reversing the comparisons.

Figure 11.1 describes the operations on a heap. The definition of a heap is an extension of a binary tree and thus inherits all of those operations as well. Note that since our implementation of a binary tree did not have any operations to add or remove elements from the tree, there are not any operations that would violate the properties of a heap. Listing 11.1 shows the interface definition for a heap. Figure 11.2 shows the UML description of the `HeapADT`.

Simply put, a minheap will always store its smallest element at the root of the binary tree, and both children of the root of a minheap are also minheaps. Figure 11.3 illustrates two valid minheaps with the same data. Let's look at the basic operations on a heap and examine generic algorithms for each.

The `addElement` Operation

The `addElement` method adds a given element to the appropriate location in the heap, maintaining both the completeness property and the ordering property of the heap. This method throws a `ClassCastException` if the given element is not `Comparable`. A binary tree is considered *complete* if it is balanced, meaning all of the leaves are at level h or h–1, where h is $\log_2 n$ and n is the number of elements in

Operation	Description
`addElement`	Adds the given element to the heap.
`removeMin`	Removes the minimum element in the heap.
`findMin`	Returns a reference to the minimum element in the heap.

FIGURE 11.1 The operations on a heap

LISTING 11.1

```
/**
 * HeapADT defines the interface to a heap.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 9/9/2008
 */

package jss2;

public interface HeapADT<T> extends BinaryTreeADT<T>
{

   /**
    * Adds the specified object to this heap.
    *
    * @param obj the element to be added to this heap
    */
   public void addElement (T obj);

   /**
    * Removes element with the lowest value from this heap.
    *
    * @return the element with the lowest value from this heap
    */
   public T removeMin();

   /**
    * Returns a reference to the element with the lowest value in
    * this heap.
    *
    * @return a reference to the element with the lowest value in this heap
    */
   public T findMin();
}
```

the tree, and all of the leaves at level h are on the left side of the tree. Because a heap is a complete tree, there is only one correct location for the insertion of a new node, and that is either the next open position from the left at level h or the first position on the left at level h + 1 if level h is full. Figure 11.4 illustrates these two possibilities.

KEY CONCEPT

The `addElement` method adds a given `Comparable` element to the appropriate location in the heap, maintaining both the completeness property and the ordering property of the heap.

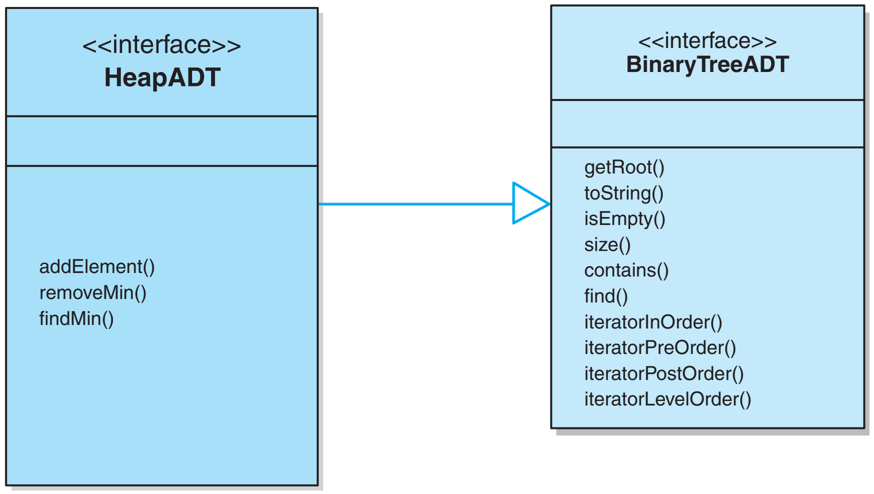


FIGURE 11.2 UML description of the `HeapADT`

KEY CONCEPT
Because a heap is a complete tree, there is only one correct location for the insertion of a new node, and that is either the next open position from the left at level h or the first position on the left at level h + 1 if level h is full.

Once we have located the new node in the proper position, we then must account for the ordering property. To do this, we simply compare the new value to its parent value and swap the values if the new node is less than its parent. We continue this process up the tree until the new value either is greater than its parent or is in the root of the heap. Figure 11.5 illustrates this process for inserting a new element into a heap. Typically, in heap implementations, we keep track of the position of the last node or, more precisely, the last leaf in the tree. After an `addElement` operation, the last node is set to the node that was inserted.

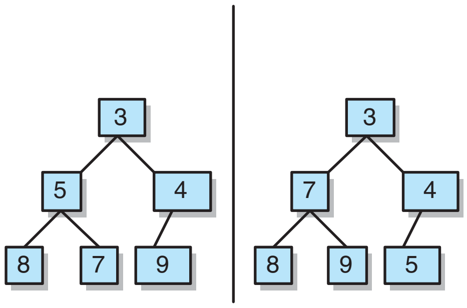


FIGURE 11.3 Two minheaps containing the same data

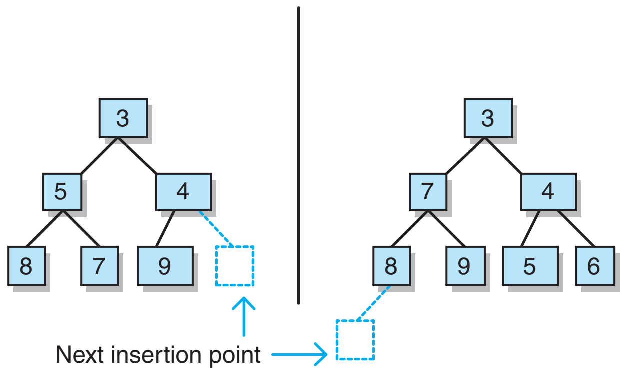


FIGURE 11.4 Insertion points for a heap

The `removeMin` Operation

KEY CONCEPT

Typically, in heap implementations, we keep track of the position of the last node or, more precisely, the last leaf in the tree.

The `removeMin` method removes the minimum element from the minheap and returns it. Because the minimum element is stored in the root of a minheap, we need to return the element stored at the root and replace it with another element in the heap. As with the `addElement` operation, to maintain the completeness of the tree, there is only one valid element to replace the root, and that is the element stored in the last leaf in the tree. This last leaf will be the rightmost leaf at level h

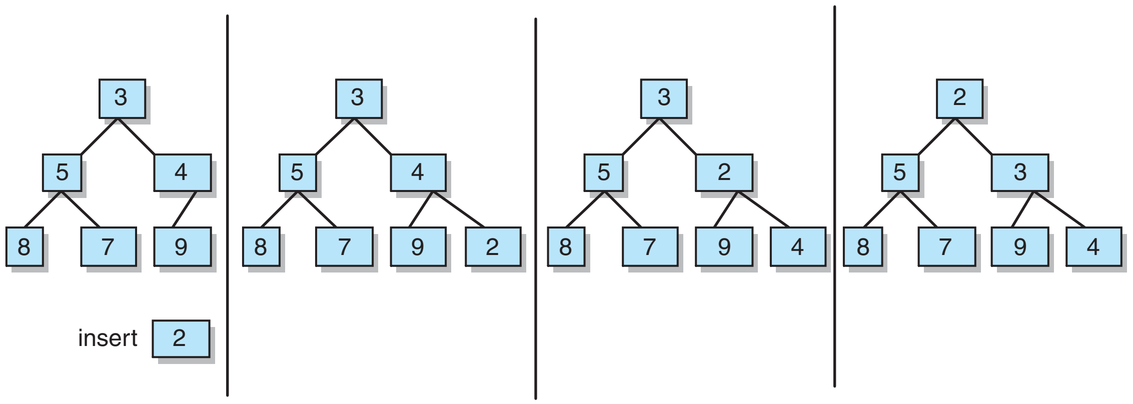


FIGURE 11.5 Insertion and reordering in a heap

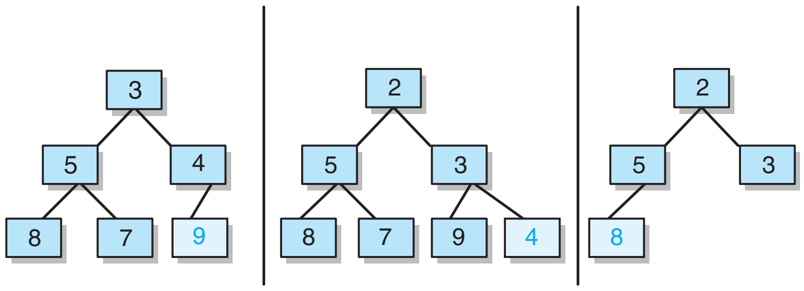


FIGURE 11.6 Examples of the last leaf in a heap

KEY CONCEPT
To maintain the completeness of the tree, there is only one valid element to replace the root, and that is the element stored in the last leaf in the tree.

of the tree. Figure 11.6 illustrates this concept of the last leaf under a variety of circumstances.

Once the element stored in the last leaf has been moved to the root, the heap will then have to be reordered to maintain the heap's ordering property. This is accomplished by comparing the new root element to the smaller of its children and then swapping them if the child is smaller. This process is repeated on down the tree until the element either is in a leaf or is less than both of its children. Figure 11.7 illustrates the process of removing the minimum element and then reordering the tree.

The `findMin` Operation

The `findMin` method returns a reference to the smallest element in the minheap. Because that element is always stored in the root of the tree, this method is simply implemented by returning the element stored in the root.

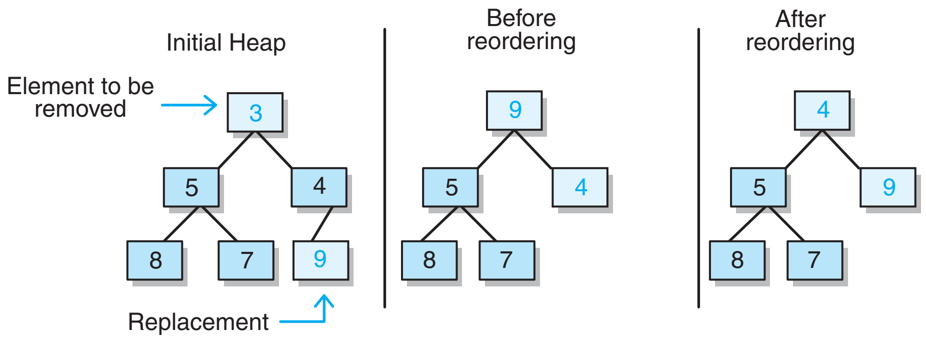


FIGURE 11.7 Removal and reordering in a heap

11.2 Using Heaps: Priority Queues

A *priority queue* is a collection that follows two ordering rules. First, items with higher priority go first. Second, items with the same priority use a first in, first out method to determine their ordering. Priority queues have a variety of applications (e.g., task scheduling in an operating system, traffic scheduling on a network, and even job scheduling at your local auto mechanic).

A priority queue could be implemented using a list of queues where each queue represents items of a given priority. Another solution to this problem is to use a minheap. Sorting the heap by priority accomplishes the first ordering (higher priority items go first). However, the first in, first out ordering of items with the same priority is something we will have to manipulate. The solution is to create a `PriorityQueueNode` object that stores the element to be placed on the queue, the priority of the element, and the order in which elements are placed on the queue. Then, we simply define the `compareTo` method for the `PriorityQueueNode` class to compare priorities first and then compare order if there is a tie. Listing 11.2 shows the `PriorityQueueNode` class, and Listing 11.3 shows the `PriorityQueue` class. The UML description of the `PriorityQueue` class is left as an exercise.

KEY CONCEPT

Though not a queue at all, a minheap provides an efficient implementation of a priority queue.

LISTING 11.2

```
/**
 * PriorityQueueNode represents a node in a priority queue containing a
 * comparable object, order, and a priority value.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 8/19/08
 */

public class PriorityQueueNode<T> implements Comparable<PriorityQueueNode>
{
   private static int nextorder = 0;
   private int priority;
   private int order;
   private T element;

   /**
    * Creates a new PriorityQueueNode with the specified data.
    *
```

LISTING 11.2 *continued*

```
    * @param obj    the element of the new priority queue node
    * @param prio   the integer priority of the new queue node
    */
   public PriorityQueueNode (T obj, int prio)
   {
      element = obj;
      priority = prio;
      order = nextorder;
      nextorder++;
   }

   /**
    * Returns the element in this node.
    *
    * @return the element contained within this node
    */
   public T getElement()
   {
      return element;
   }

   /**
    * Returns the priority value for this node.
    *
    * @return the integer priority for this node
    */
   public int getPriority()
   {
      return priority;
   }

   /**
    * Returns the order for this node.
    *
    * @return the integer order for this node
    */
   public int getOrder()
   {
      return order;
   }

   /**
    * Returns a string representation for this node.
    *
    */
```

LISTING 11.2 *continued*

```
   public String toString()
   {
      String temp = (element.toString() + priority + order);
      return temp;
   }

   /**
    * Returns the 1 if the current node has higher priority than
    * the given node and -1 otherwise.
    *
    * @param obj   the node to compare to this node
    * @return      the integer result of the comparison of the obj node and
    *              this one
    */
   public int compareTo(PriorityQueueNode obj)
   {
      int result;
      PriorityQueueNode<T> temp = obj;
      if (priority > temp.getPriority())
         result = 1;
      else if (priority < temp.getPriority())
         result = -1;
      else if (order > temp.getOrder())
         result = 1;
      else
         result = -1;

      return result;
   }
}
```

LISTING 11.3

```
/**
 * PriorityQueue demonstrates a priority queue using a Heap.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
```

LISTING 11.3 *continued*

```
 * @version 1.0, 8/19/08
 */

import jss2.*;

public class PriorityQueue<T> extends ArrayHeap<PriorityQueueNode<T>>
{

   /**
    * Creates an empty priority queue.
    */
   public PriorityQueue()
   {
      super();
   }

   /**
    * Adds the given element to this PriorityQueue.
    *
    * @param object the element to be added to the priority queue
    * @param priority the integer priority of the element to be added
    */
   public void addElement (T object, int priority)
   {
      PriorityQueueNode<T> node = new PriorityQueueNode<T> (object, priority);
         super.addElement(node);
   }

   /**
    * Removes the next highest priority element from this priority
    * queue and returns a reference to it.
    *
    * @return a reference to the next highest priority element in this queue
    */
   public T removeNext()
   {
      PriorityQueueNode<T> temp = (PriorityQueueNode<T>)super.removeMin();
      return temp.getElement();
   }
}
```

11.3 Implementing Heaps: With Links

All of our implementations of trees thus far have been illustrated using links. Thus it is natural to extend that discussion to a linked implementation of a heap. Because of the requirement that we be able to traverse up the tree after an insertion, it is necessary for the nodes in a heap to store a pointer to their parent. Because our `BinaryTreeNode` class did not have a parent pointer, we start our linked implementation by creating a `HeapNode` class that extends our `BinaryTreeNode` class and adds a parent pointer. Listing 11.4 shows the `HeapNode` class.

KEY CONCEPT

Because of the requirement that we be able to traverse up the tree after an insertion, it is necessary for the nodes in a heap to store a pointer to their parent.

The additional instance data for a linked implementation will consist of a single reference to a `HeapNode` called `lastNode` so that we can keep track of the last leaf in the heap:

```
public HeapNode lastNode;
```

The `addElement` Operation

The `addElement` method must accomplish three tasks: add the new node at the appropriate location, reorder the heap to maintain the ordering property, and then reset the `lastNode` pointer to point to the new last node.

LISTING 11.4

```
/**
 * HeapNode creates a binary tree node with a parent pointer for use
 * in heaps.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 9/9/2008
 */

package jss2;

public class HeapNode<T> extends BinaryTreeNode<T>
{
   protected HeapNode<T> parent;

   /**
    * Creates a new heap node with the specified data.
    *
    * @param obj the data to be contained within the new heap nodes
    */
```

LISTING 11.4 *continued*

```
      HeapNode (T obj)
      {
         super(obj);
         parent = null;
      }
   }
```

```
   /**
    * Adds the specified element to this heap in the appropriate
    * position according to its key value. Note that equal elements
    * are added to the right.
    *
    * @param obj the element to be added to this heap
    */
   public void addElement (T obj)
   {
      HeapNode<T> node = new HeapNode<T>(obj);

      if (root == null)
         root=node;
      else
      {
         HeapNode<T> next_parent = getNextParentAdd();
         if (next_parent.left == null)
            next_parent.left = node;
         else
            next_parent.right = node;

         node.parent = next_parent;
      }
      lastNode = node;
      count++;
      if (count>1)
         heapifyAdd();
   }
```

This method also uses two private methods: getNextParentAdd, which returns a reference to the node that will be the parent of the node to be inserted, and heapifyAdd, which accomplishes any necessary reordering of the heap starting with the new leaf and working up toward the root. Both of those methods are shown below.

```
/**
 * Returns the node that will be the parent of the new node
 *
 * @return the node that will be a parent of the new node
 */
private HeapNode<T> getNextParentAdd()
{
   HeapNode<T> result = lastNode;

   while ((result != root) && (result.parent.left != result))
      result = result.parent;

   if (result != root)
      if (result.parent.right == null)
         result = result.parent;
      else
      {
         result = (HeapNode<T>)result.parent.right;
         while (result.left != null)
            result = (HeapNode<T>)result.left;
      }
   else
      while (result.left != null)
         result = (HeapNode<T>)result.left;

   return result;
}
```

```
/**
 * Reorders this heap after adding a node.
 */
private void heapifyAdd()
{
   T temp;
   HeapNode<T> next = lastNode;
```

```
      temp = next.element;

      while ((next != root) && (((Comparable)temp).compareTo
                                 (next.parent.element) < 0))
      {
         next.element = next.parent.element;
         next = next.parent;
      }
      next.element = temp;
   }
```

In this linked implementation, the first step in the process of adding an element is to determine the parent of the node to be inserted. Because, in the worst case, this involves traversing from the bottom-right node of the heap up to the root and then down to the bottom-left node of the heap, this step has time complexity $2 \times \log n$. The next step is to insert the new node. Because this involves only simple assignment statements, this step has constant time complexity (O(1)). The last step is to reorder the path from the inserted leaf to the root if necessary. This process involves at most log n comparisons because that is the length of the path. Thus the `addElement` operation for the linked implementation has time complexity $2 \times \log n + 1 + \log n$ or O(log n).

Note that the `heapifyAdd` method does not perform a full swap of parent and child as it moves up the heap. Instead, it simply shifts parent elements down until a proper insertion point is found and then assigns the new value into that location. This does not actually improve the O() of the algorithm as it would be O(log n) even if we were performing full swaps. However, it does improve the efficiency since it reduces the number of assignments performed at each level of the heap.

The `removeMin` Operation

The `removeMin` method must accomplish three tasks: replace the element stored in the root with the element stored in the last node, reorder the heap if necessary, and return the original root element. Like the `addElement` method, the `removeMin` method uses two additional methods: `getNewLastNode`, which returns a reference to the node that will be the new last node, and `heapifyRemove`, which will accomplish any necessary reordering of the tree starting from the root down. All three of these methods are shown below.

```
/**
 * Remove the element with the lowest value in this heap and
 * returns a reference to it. Throws an EmptyCollectionException
 * if the heap is empty.
 *
 * @return                              the element with the lowest value in
 *                                      this heap
 * @throws EmptyCollectionException     if an empty collection exception
 *                                      occurs
 */
public T removeMin() throws EmptyCollectionException
{
   if (isEmpty())
      throw new EmptyCollectionException ("Empty Heap");

   T minElement = root.element;

   if (count == 1)
   {
      root = null;
      lastNode = null;
   }
   else
   {
      HeapNode<T> next_last = getNewLastNode();
      if (lastNode.parent.left == lastNode)
         lastNode.parent.left = null;
      else
         lastNode.parent.right = null;

      root.element = lastNode.element;
      lastNode = next_last;
      heapifyRemove();
   }

   count--;

   return minElement;
}
```

```
/**
 * Returns the node that will be the new last node after a remove.
 *
 * @return the node that will be the new last node after a remove
 */
private HeapNode<T> getNewLastNode()
{
   HeapNode<T> result = lastNode;

   while ((result != root) && (result.parent.left == result))
      result = result.parent;

   if (result != root)
      result = (HeapNode<T>)result.parent.left;

   while (result.right != null)
      result = (HeapNode<T>)result.right;

   return result;
}
```

```
/**
 * Reorders this heap after removing the root element.
 */
private void heapifyRemove()
{
   T temp;
   HeapNode<T> node = (HeapNode<T>)root;
   HeapNode<T> left = (HeapNode<T>)node.left;
   HeapNode<T> right = (HeapNode<T>)node.right;
   HeapNode<T> next;

   if ((left == null) && (right == null))
      next = null;
   else if (left == null)
      next = right;
   else if (right == null)
      next = left;
   else if (((Comparable)left.element).compareTo(right.element) < 0)
      next = left;
      else
         next = right;

      temp = node.element;
      while ((next != null) && (((Comparable)next.element).compareTo
                                 (temp) < 0))
      {
         node.element = next.element;
         node = next;
         left = (HeapNode<T>)node.left;
         right = (HeapNode<T>)node.right;

         if ((left == null) && (right == null))
            next = null;
         else if (left == null)
            next = right;
         else if (right == null)
            next = left;
         else if (((Comparable)left.element).compareTo(right.element) < 0)
            next = left;
         else
            next = right;
      }
      node.element = temp;
   }
```

The `removeMin` method for the linked implementation must remove the root element and replace it with the element from the last node. Because this is simply assignment statements, this step has time complexity 1. Next, this method must reorder the heap if necessary from the root down to a leaf. Because the maximum path length from the root to a leaf is log n, this step has time complexity log n. Finally, we must determine the new last node. Like the process for determining the next parent node for the `addElement` method, the worst case is that we must traverse from a leaf through the root and down to another leaf. Thus the time complexity of this step is 2*log n. The resulting time complexity of the `removeMin` operation is 2*log n + log n + 1 or O(log n).

The `findMin` Operation

The `findMin` method simply returns a reference to the element stored at the root of the heap and therefore is O(1).

11.4 Implementing Heaps: With Arrays

An array implementation of a heap may provide a simpler alternative than our linked implementation. Many of the intricacies of the linked implementation relate to the need to traverse up and down the tree to determine the last leaf of the tree or to determine the parent of the next node to insert. Many of those difficulties do not exist in the array implementation because we are able to determine the last node in the tree by looking at the last element stored in the array.

KEY CONCEPT

In an array implementation of a binary tree, the root of the tree is in position 0, and for each node n, n's left child is in position 2n + 1 and n's right child is in position 2(n + 1).

As we discussed in Chapter 9, a simple array implementation of a binary tree can be created using the notion that the root of the tree is in position 0, and for each node n, n's left child will be in position 2n+1 of the array and n's right child will be in position 2(n + 1) of the array. Of course, the inverse is also true. For any node n other than the root, n's parent is in position (n – 1)/2. Because of our ability to calculate the location of both parent and child, unlike the linked implementation, the array implementation does not require the creation of a `HeapNode` class. The UML description of the array implementation of a heap is left as an exercise.

The `addElement` Operation

The `addElement` method for the array implementation must accomplish three tasks: add the new node at the appropriate location, reorder the heap to maintain the ordering property, and increment the count by one. Of course, as with all of our array implementations, the method must first check for available space and expand the capacity of the array if necessary. Like the linked implementation, the `addElement` operation of the array implementation uses a private method called `heapifyAdd` to reorder the heap if necessary.

```
/**
 * Adds the specified element to this heap in the appropriate
 * position according to its key value. Note that equal elements
 * are added to the right.
 *
 * @param obj the element to be added to this heap
 */
public void addElement (T obj)
{
      if (count==tree.length)
         expandCapacity();

      tree[count] =obj;
      count++;

      if (count>1)
         heapifyAdd();
   }
```

```
   /**
    * Reorders this heap to maintain the ordering property after
    * adding a node.
    */
   private void heapifyAdd()
   {
      T temp;
      int next = count - 1;

      temp = tree[next];

      while ((next != 0) && (((Comparable)temp).compareTo
                                 (tree[(next-1)/2]) < 0))
      {

         tree[next] = tree[(next-1)/2];
         next = (next-1)/2;
      }

      tree[next] = temp;
   }
```

Unlike the linked implementation, the array implementation does not require the first step of determining the parent of the new node. However, both of the other steps are the same as those for the linked implementation. Thus the time complexity for the `addElement` operation for the array implementation is 1 + log n or O(log n). Granted, the two implementations have the same Order(), but the array implementation is more efficient.

KEY CONCEPT
The `addElement` operation for both the linked and array implementations is O(log n).

The removeMin Operation

The removeMin method must accomplish three tasks: replace the element stored in the root with the element stored in the last element, reorder the heap if necessary, and return the original root element. In the case of the array implementation, we know the last element of the heap is stored in position count−1 of the array. We then use a private method heapifyRemove to reorder the heap as necessary.

```
/**
 * Remove the element with the lowest value in this heap and
 * returns a reference to it. Throws an EmptyHeapException if
 * the heap is empty.
 *
 * @return                              a reference to the element with the
 *                                      lowest value in this head
 * @throws EmptyCollection Exception    if an empty collection exception
 *                                      occurs
 */
public T removeMin() throws EmptyCollectionException
{
   if (isEmpty())
      throw new EmptyCollectionException ("Empty Heap");

   T minElement = tree[0];
   tree[0] = tree[count-1];
   heapifyRemove();
   count--;

   return minElement;
}
```

```
/**
 * Reorders this heap to maintain the ordering property.
 */
private void heapifyRemove()
{
   T temp;
   int node = 0;
   int left = 1;
   int right = 2;
   int next;
```

```
      if ((tree[left] == null) && (tree[right] == null))
         next = count;
      else if (tree[left] == null)
         next = right;
      else if (tree[right] == null)
         next = left;
      else if (((Comparable)tree[left]).compareTo(tree[right]) < 0)
         next = left;
      else
         next = right;
      temp = tree[node];

      while ((next < count) && (((Comparable)tree[next]).compareTo
                                  (temp) < 0))
      {
         tree[node] = tree[next];
         node = next;
         left = 2*node+1;
         right = 2*(node+1);
         if ((tree[left] == null) && (tree[right] == null))
            next = count;
         else if (tree[left] == null)
            next = right;
         else if (tree[right] == null)
            next = left;
         else if (((Comparable)tree[left]).compareTo(tree[right]) < 0)
            next = left;
         else
            next = right;
      }
      tree[node] = temp;
   }
```

Like the `addElement` method, the array implementation of the `removeMin` operation looks just like the linked implementation except that it does not have to determine the new last node. Thus the resulting time complexity is log n + 1 or O(log n).

KEY CONCEPT

The `removeMin` operation for both the linked and array implementations is O(log n).

The `findMin` Operation

Like the linked implementation, the `findMin` method simply returns a reference to the element stored at the root of the heap or position 0 of the array and therefore is O(1).

11.5 Using Heaps: Heap Sort

In Chapter 8, we introduced a variety of sorting techniques, some of which were sequential sorts (bubble sort, selection sort, and insertion sort) and some of which were logarithmic sorts (merge sort and quick sort). In that chapter, we also introduced a queue-based sort called a radix sort. Given the ordering property of a heap, it is natural to think of using a heap to sort a list of numbers. The process is quite simple. Simply add each of the elements of the list to a heap and then remove them one at a time from the root. In the case of a minheap, the result will be the list in ascending order. In the case of a maxheap, the result will be the list in descending order. Because both the `add` and `remove` operations are O(log n), it might be tempting to conclude that a heap sort is also O(log n). However, keep in mind that those operations are O(log n) to add or remove a single element in a list of n elements. Insertion into a heap is O(log n) for any given node, and thus would be O(n log n) for n nodes. Removal is also O(log n) for a single node and thus O(n log n) for n nodes. With the heap sort algorithm, we are performing both operations, `addElement` and `removeMin`, n times, once for each of the elements in the list. Therefore, the resulting time complexity is 2 × n × log n or O(n log n).

KEY CONCEPT
The `heapSort` method consists of adding each of the elements of the list to a heap and then removing them one at a time.

It is also possible to "build" a heap in place using the array to be sorted. Because we know the relative position of each parent and child in the heap, we can simply start with the first non-leaf node in the array, compare it to its children, and swap if necessary. We then work backward in the array until we reach the root. Because, at most, this will require us to make two comparisons for each non-leaf node, this approach is O(n) to build the heap. However, using this approach, removing each element from the heap and maintaining the properties of the heap would still be O(n log n). Thus, even though this approach is slightly more efficient, roughly 2 × n + n log n, it is still O(n log n). The implementation of this approach is left as an exercise. The `heapSort` method could be added to our class of search and sort methods described in Chapter 8. Listing 11.5 illustrates how it might be created as a standalone class.

KEY CONCEPT
Heap sort is O(n log n).

LISTING 11.5

```
/**
 * HeapSort sorts a given array of Comparable objects using a heap.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 8/19/08
 */
```

LISTING 11.5 *continued*

```
package jss2;

public class HeapSort<T>
{

   /**
    * @param data   the data to be added to the heapsort
    * @param min    the integer minimum value
    * @param max    the integer maximum value
    */
   public void HeapSort(T[] data, int min, int max)
   {
      ArrayHeap<T> temp = new ArrayHeap<T>();

      /** copy the array into a heap */
      for (int ct = min; ct <= max; ct++)
         temp.addElement(data[ct]);

      /** place the sorted elements back into the array */
      int count = min;
      while (!(temp.isEmpty()))
      {
         data[count] = temp.removeMin();
         count++;
      }
   }
}
```

Summary of Key Concepts

- A minheap is a complete binary tree in which each node is less than or equal to both the left child and the right child.
- A minheap stores its smallest element at the root of the binary tree, and both children of the root of a minheap are also minheaps.
- The `addElement` method adds a given `Comparable` element to the appropriate location in the heap, maintaining both the completeness property and the ordering property of the heap.
- Because a heap is a complete tree, there is only one correct location for the insertion of a new node, and that is either the next open position from the left at level h or the first position on the left at level h + 1 if level h is full.
- Typically, in heap implementations, we keep track of the position of the last node or, more precisely, the last leaf in the tree.
- To maintain the completeness of the tree, there is only one valid element to replace the root, and that is the element stored in the last leaf in the tree.
- Though not a queue at all, a minheap provides an efficient implementation of a priority queue.
- Because of the requirement that we be able to traverse up the tree after an insertion, it is necessary for the nodes in a heap to store a pointer to their parent.
- In an array implementation of a binary tree, the root of the tree is in position 0, and for each node n, n's left child is in position 2n + 1 and n's right child is in position 2(n + 1).
- The `addElement` operation for both the linked and array implementations is O(log n).
- The `removeMin` operation for both the linked and array implementations is O(log n).
- The `heapSort` method consists of adding each of the elements of the list to a heap and then removing them one at a time.
- Heap sort is O(n log n).

Self-Review Questions

SR 11.1 What is the difference between a heap (a minheap) and a binary search tree?

SR 11.2 What is the difference between a minheap and a maxheap?

SR 11.3 What does it mean for a heap to be complete?

SR 11.4 Does a heap ever have to be rebalanced?

SR 11.5 The `addElement` operation for the linked implementation must determine the parent of the next node to be inserted. Why?

SR 11.6 Why does the `addElement` operation for the array implementation not have to determine the parent of the next node to be inserted?

SR 11.7 The `removeMin` operation for both implementations replaces the element at the root with the element in the last leaf of the heap. Why is this the proper replacement?

SR 11.8 What is the time complexity of the `addElement` operation?

SR 11.9 What is the time complexity of the `removeMin` operation?

SR 11.10 What is the time complexity of heap sort?

Exercises

EX 11.1 Draw the heap that results from adding the following integers (34 45 3 87 65 32 1 12 17).

EX 11.2 Starting with the resulting tree from Exercise 11.1, draw the tree that results from performing a `removeMin` operation.

EX 11.3 Starting with an empty minheap, draw the heap after each of the following operations:

```
addElement(40);
addElement(25):
removeMin();
addElement(10);
removeMin();
addElement(5);
addElement(1);
removeMin();
addElement(45);
addElement(50);
```

EX 11.4 Repeat Exercise 11.3, this time with maxheap.

EX 11.5 Draw the UML description for the `PriorityQueue` class described in this chapter.

EX 11.6 Draw the UML description for the array implementation of heap described in this chapter.

Programming Projects

PP 11.1 Implement a queue using a heap. Keep in mind that a queue is a first in, first out structure. Thus the comparison in the heap will have to be according to order entry into the queue.

PP 11.2 Implement a stack using a heap. Keep in mind that a stack is a last in, first out structure. Thus the comparison in the heap will have to be according to order entry into the queue.

PP 11.3 Implement a maxheap using an array implementation.

PP 11.4 Implement a maxheap using a linked implementation.

PP 11.5 It is possible to make the heap sort algorithm more efficient by writing a method that will order the entire list at once instead of adding the elements one at a time. Implement such a method, and rewrite the heap sort algorithm to make use of it.

PP 11.6 Use a heap to implement a simulator for a process scheduling system. In this system, jobs will be read from a file consisting of the job id (a six-character string), the length of the job (an `int` representing seconds), and the priority of the job (an `int` where the higher the number the higher the priority). Each job will also be assigned an arrival number (an `int` representing the order of its arrival). The simulation should output the job id, the priority, the length of the job, and the completion time (relative to a simulation start time of 0).

PP 11.7 Create a birthday reminder system using a minheap such that the ordering on the heap is done each day according to days remaining until the individual's birthday. Keep in mind that when a birthday passes, the heap must be reordered.

PP 11.8 In Section 11.2, we described a more efficient heap sort algorithm that would build the heap within the existing array. Implement this more efficient heap sort algorithm.

Answers to Self-Review Questions

SRA 11.1 A binary search tree has the ordering property that the left child of any node is less than the node, and the node is less than or equal to its right child. A minheap is complete and has the ordering property that the node is less than both of its children.

SRA 11.2 A minheap has the ordering property that the node is less than both of its children. A maxheap has the ordering property that the node is greater than both of its children.

SRA 11.3 A heap is considered complete if it is balanced, meaning all of the leaves are at level h or h – 1, where h is $\log_2 n$ and n is the number of elements in the tree, and all of the leaves at level h are on the left side of the tree.

SRA 11.4 No. By definition, a complete heap is balanced and the algorithms for `add` and `remove` maintain that balance.

SRA 11.5 The `addElement` operation must determine the parent of the node to be inserted so that a child pointer of that node can be set to the new node.

SRA 11.6 The `addElement` operation for the array implementation does not have to determine the parent of the new node because the new element is inserted in position `count` of the array and its parent is determined by position in the array.

SRA 11.7 To maintain the completeness of the tree, the only valid replacement for the element at the root is the element at the last leaf. Then the heap must be reordered as necessary to maintain the ordering property.

SRA 11.8 For both implementations, the `addElement` operation is O(log n). However, despite having the same order, the array implementation is somewhat more efficient because it does not have to determine the parent of the node to be inserted.

SRA 11.9 For both implementations, the `removeMin` operation is O(log n). However, despite having the same order, the array implementation is somewhat more efficient because it does not have to determine the new last leaf.

SRA 11.10 The heap sort algorithm is O(n log n).

This page intentionally left blank

12 Multi-way Search Trees

When we first introduced the concept of efficiency of algorithms, we said that we were interested in issues such as processing time and memory. In this chapter, we explore multi-way trees that were specifically designed with a concern for the use of space and the effect that a particular use of space could have on the total processing time for an algorithm.

CHAPTER OBJECTIVES

- Examine 2-3 and 2-4 trees
- Introduce the generic concept of a B-tree
- Examine some specialized implementations of B-trees

12.1 Combining Tree Concepts

In Chapter 9, we established the difference between a general tree, which has a varying number of children per node, and a binary tree, which has at most two children per node. Then in Chapter 10, we discussed the concept of a search tree, which has a specific ordering relationship among the elements in the nodes to allow efficient searching for a target value. In particular, we focused on binary search trees. Now we can combine these concepts and extend them further.

KEY CONCEPT
A multi-way search tree can have more than two children per node and can store more than one element in each node.

In a *multi-way search tree*, each node might have more than two child nodes, and, because it is a search tree, there is a specific ordering relationship among the elements. Furthermore, a single node in a multi-way search tree may store more than one element.

This chapter examines three specific forms of a multi-way search tree:

- 2-3 trees
- 2-4 trees
- B-trees

12.2 2-3 Trees

A *2-3 tree* is a multi-way search tree in which each node has two children (referred to as a *2-node*) or three children (referred to as a *3-node*). A 2-node contains one element and, like a binary search tree, the left subtree contains elements that are less than that element and the right subtree contains elements that are greater than or equal to that element. However, unlike a binary search tree, a 2-node can have either no children or two children—it cannot have just one child.

A 3-node contains two elements, one designated as the smaller element and one designated as the larger element. A 3-node has either no children or three children. If a 3-node has children, the left subtree contains elements that are less than the smaller element and the right subtree contains elements that are greater than or equal to the larger element. The middle subtree contains elements that are greater than or equal to the smaller element and less than the larger element.

KEY CONCEPT
A 2-3 tree contains nodes that contain either one or two elements and have either zero, two, or three children.

All of the leaves of a 2-3 tree are on the same level. Figure 12.1 illustrates a valid 2-3 tree.

Inserting Elements into a 2-3 Tree

Similar to a binary search tree, all insertions into a 2-3 tree occur at the leaves of the tree. That is, the tree is searched to determine where the new element will go;

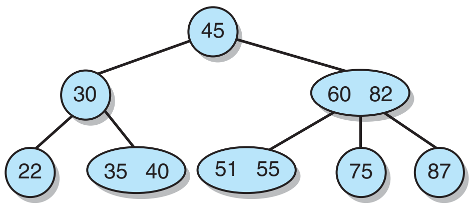


FIGURE 12.1 A 2-3 tree

then it is inserted. Unlike a binary tree, however, the process of inserting an element into a 2-3 tree can have a ripple effect on the structure of the rest of the tree.

Inserting an element into a 2-3 tree has three cases. The first, and simplest, case is that the tree is empty. In this case, a new node is created containing the new element, and this node is designated as the root of the tree.

The second case occurs when we want to insert a new element at a leaf that is a 2-node. That is, we traverse the tree to the appropriate leaf (which may also be the root) and find that the leaf is a 2-node (containing only one element). In this case, the new element is added to the 2-node, making it a 3-node. Note that the new element may be less than or greater than the existing element. Figure 12.2 illustrates this case by inserting the value 27 into the tree pictured in Figure 12.1. The leaf node containing 22 is a 2-node, therefore 27 is inserted into that node, making it a 3-node. Note that neither the number of nodes in the tree nor the height of the tree changed because of this insertion.

The third insertion situation occurs when we want to insert a new element at a leaf that is a 3-node (containing two elements). In this case, because the 3-node cannot hold any more elements, it is split, and the middle element is moved up a level in the tree. The middle element that moves up a level could be either of the two elements that already existed in the 3-node, or it could be the new element being inserted. It depends on the relationship among those three elements.

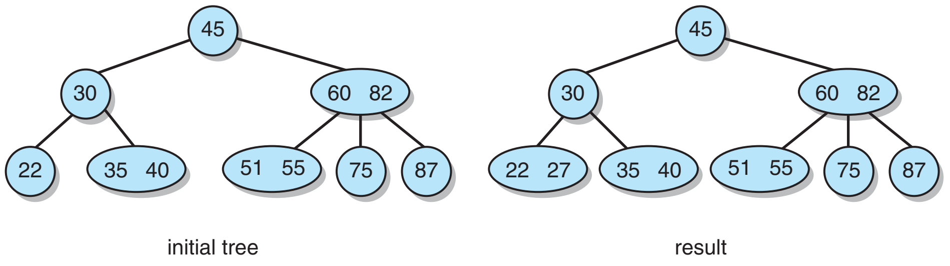


FIGURE 12.2 Inserting 27

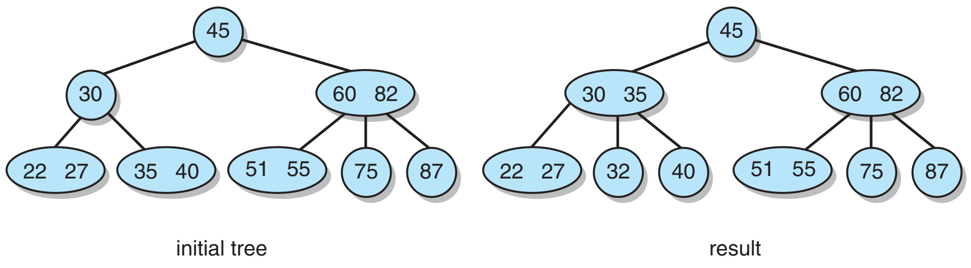


FIGURE 12.3 Inserting 32

Figure 12.3 shows the result of inserting the element 32 into the tree from Figure 12.2. Searching the tree, we reach the 3-node that contains the elements 35 and 40. That node is split, and the middle element (35) is moved up to join its parent node. Thus the internal node that contains 30 becomes a 3-node that contains both 30 and 35. Note that the act of splitting a 3-node results in two 2-nodes at the leaf level. In this example, we are left with one 2-node that contains 32 and another 2-node that contains 40.

Now consider the situation in which we must split a 3-node whose parent is already a 3-node. The middle element that is promoted causes the parent to split, moving an element up yet another level in the tree. Figure 12.4 shows the effect of inserting the element 57 into the tree from Figure 12.3. Searching the tree, we reach the 3-node leaf that contains 51 and 55. This node is split, causing the middle element 55 to move up a level. But that node is already a 3-node, containing the values 60 and 82, so we split that node as well, promoting the element 60, which joins the 2-node containing 45 at the root. Therefore, inserting an element into a 2-3 tree can cause a ripple effect that changes several nodes in the tree.

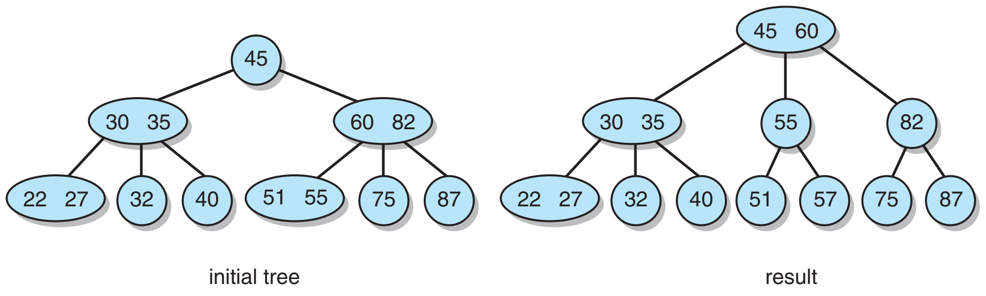


FIGURE 12.4 Inserting 57

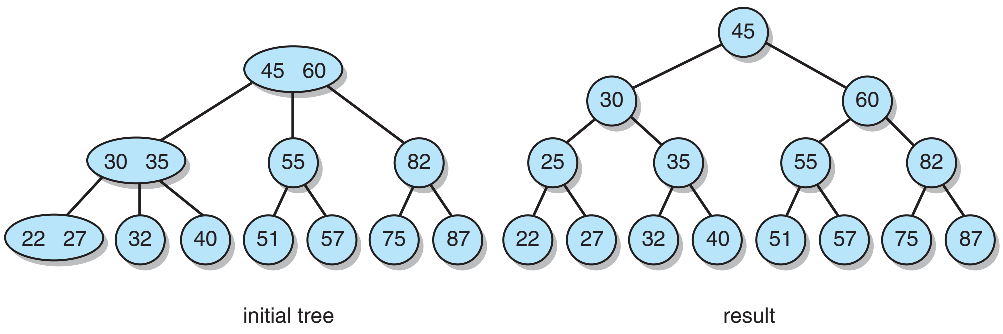


FIGURE 12.5 Inserting 25

If this effect propagates all the way to the root of the entire tree, a new 2-node root is created. For example, inserting the element 25 into the tree from Figure 12.4 results in the tree depicted in Figure 12.5. The 3-node containing 22 and 27 is split, promoting 25. This causes the 3-node containing 30 and 35 to split, promoting 30. This causes the 3-node containing 45 and 60 (which happens to be the root of the entire tree) to split, creating a new 2-node root that contains 45.

Note that when the root of the tree splits, the height of the tree increases by one. The insertion strategy for a 2-3 tree keeps all of the leaves at the same level.

KEY CONCEPT

If the propagation effect of a 2-3 tree insertion causes the root to split, the tree increases in height.

Removing Elements from a 2-3 Tree

Removal of elements from a 2-3 tree is also made up of three cases. The first case is that the element to be removed is in a leaf that is a 3-node. In this case, removal is simply a matter of removing the element from the node. Figure 12.6 illustrates

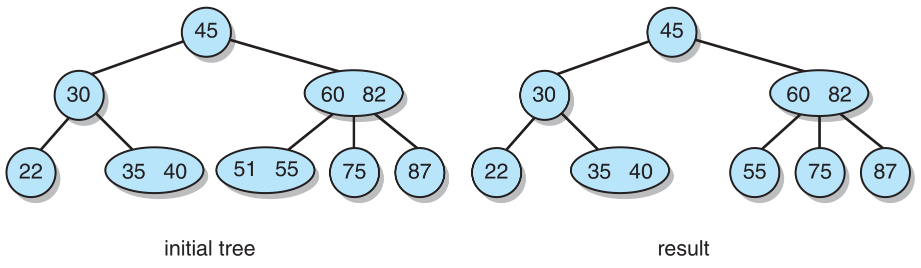


FIGURE 12.6 Removal from a 2-3 tree (case 1)

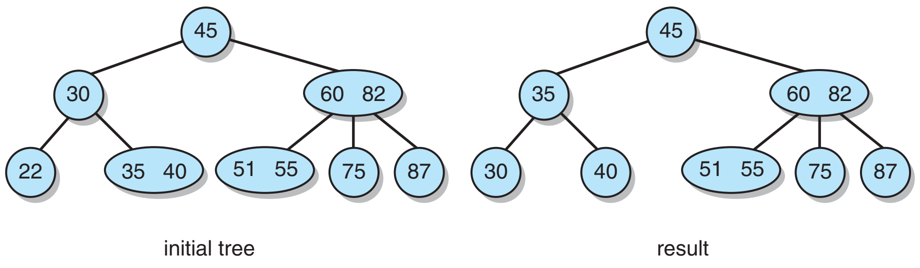


FIGURE 12.7 Removal from a 2-3 tree (case 2.1)

this process by removing the element 51 from the tree we began with in Figure 12.1. Note that the properties of a 2-3 tree are maintained.

The second case is that the element to be removed is in a leaf that is a 2-node. This condition is called *underflow* and creates a situation in which we must rotate the tree and/or reduce the tree's height in order to maintain the properties of the 2-3 tree. This situation can be broken down into four subordinate cases that we will refer to as cases 2.1, 2.2, 2.3, and 2.4. Figure 12.7 illustrates case 2.1 and shows what happens if we remove the element 22 from our initial tree shown in Figure 12.1. In this case, because the parent node has a right child that is a 3-node, we can maintain the properties of a 2-3 tree by rotating the smaller element of the 3-node around the parent. The same process will work if the element being removed from a 2-node leaf is the right child and the left child is a 3-node.

What happens if we now remove the element 30 from the resulting tree in Figure 12.7? We can no longer maintain the properties of a 2-3 tree through a local rotation. Keep in mind, a node in a 2-3 tree cannot have just one child. Because the leftmost child of the right child of the root is a 3-node, we can rotate the smaller element of that node around the root to maintain the properties of a 2-3 tree. This process is illustrated in Figure 12.8 and represents case 2.2. Notice that the element 51 moves to the root, the element 45 becomes the larger element in a 3-node leaf, and then the smaller element of that leaf is rotated around its parent. Once element 51 was moved to the root and element 45 was moved to a 3-node leaf, we were back in the same situation as case 2.1.

Given the resulting 2-3 tree in Figure 12.8, what happens if we now remove element 55? None of the leaves of this tree are 3-nodes. Thus, rotation from a leaf, even from a distance, is no longer an option. However, because the parent node is a 3-node, all that is required to maintain the properties of a 2-3 node is to change this 3-node to a 2-node by rotating the smaller element (60) into what will now be the left child of the node. Figure 12.9 illustrates case 2.3.

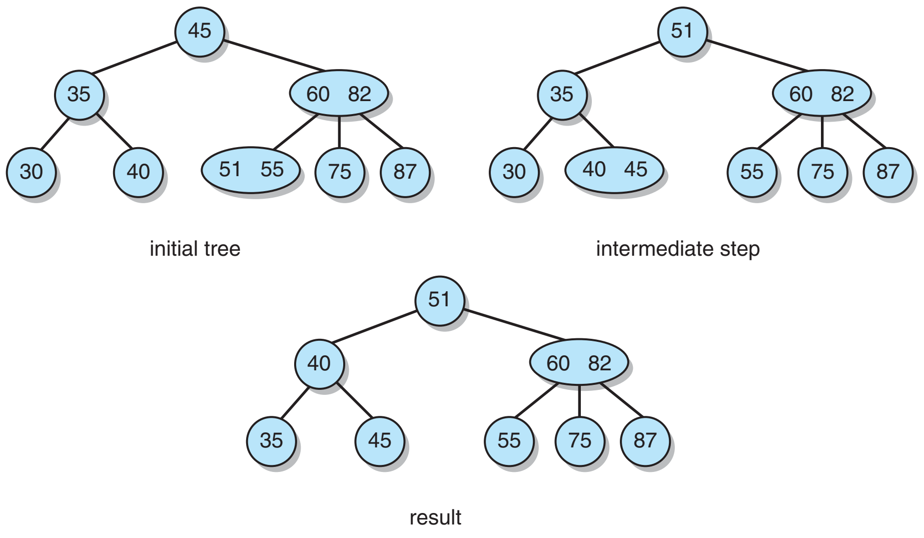


FIGURE 12.8 Removal from a 2-3 tree (case 2.2)

If we then remove element 60 (using case 1), the resulting tree contains nothing but 2-nodes. Now, if we remove another element, perhaps element 45, rotation is no longer an option. We must instead reduce the height of the tree in order to maintain the properties of a 2-3 tree. This is case 2.4. To accomplish this, we simply combine each of the leaves with their parent and siblings in order. If any of these combinations contains more than two elements, we split it into two 2-nodes and promote or propagate the middle element. Figure 12.10 illustrates this process for reducing the height of the tree.

The third case is that the element to be removed is in an internal node. As we did with binary search trees, we can simply replace the element to be removed

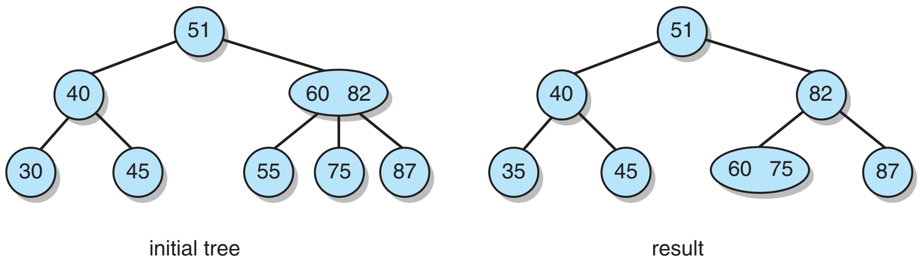


FIGURE 12.9 Removal from a 2-3 tree (case 2.3)

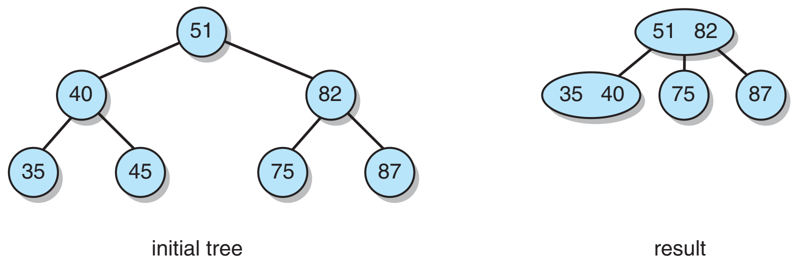


FIGURE 12.10 Removal from a 2-3 tree (case 2.4)

with its inorder successor. In a 2-3 tree, the inorder successor of an internal element will always be a leaf, which, if it is a 2-node, will bring us back to our first case, and if it is a 3-node, requires no further action. Figure 12.11 illustrates these possibilities by removing the element 30 from our original tree from Figure 12.1 and then by removing the element 60 from the resulting tree.

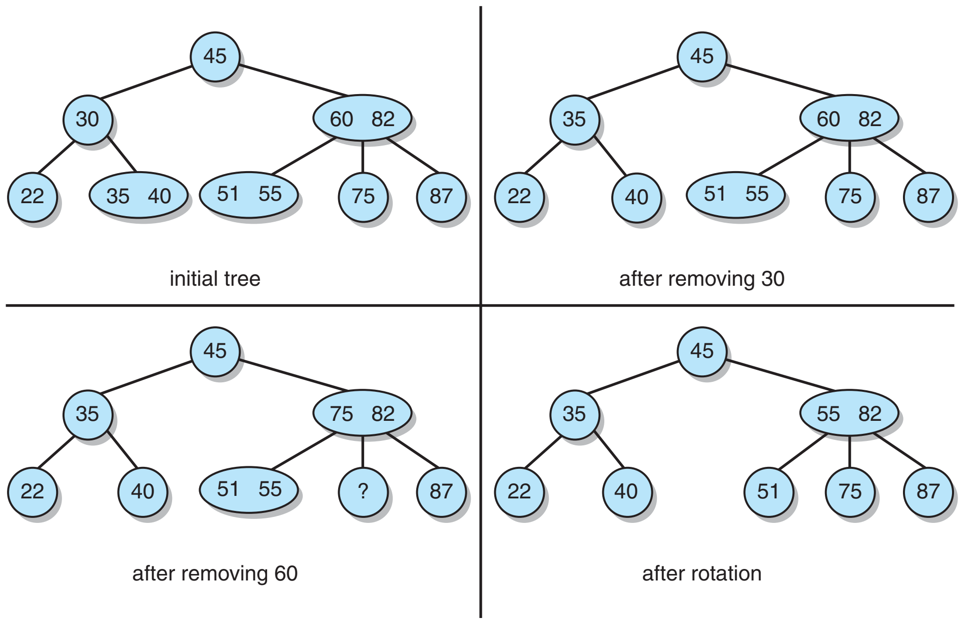


FIGURE 12.11 Removal from a 2-3 tree (case 3)

12.3 2-4 Trees

A *2-4 tree* is similar to a 2-3 tree, adding the characteristic that a node can contain three elements. Expanding on the same principles as a 2-3 tree, a *4-node* contains three elements and has either no children or four children. The same ordering property applies: the left child will be less than the leftmost element of a node, which will be less than or equal to the second child of the node, which will be less than the second element of the node, which will be less than or equal to the third child of the node, which will be less than the third element of the node, which will be less than or equal to the fourth child of the node.

KEY CONCEPT

A 2-4 tree expands on the concept of a 2-3 tree to include the use of 4-nodes.

The same cases for insertion and removal of elements apply, with 2-nodes and 3-nodes behaving similarly on insertion and 3-nodes and 4-nodes behaving similarly on removal. Figure 12.12 illustrates a series of insertions into a 2-4 tree. Figure 12.13 illustrates a series of removals from a 2-4 tree.

12.4 B-Trees

Both 2-3 and 2-4 trees are examples of a larger class of multi-way search trees called *B-trees*. We refer to the maximum number of children of each node as the *order* of the B-tree. Thus, 2-3 trees are order 3 B-trees, and 2-4 trees are order 4 B-trees.

KEY CONCEPT

A B-tree extends the concept of 2-3 and 2-4 trees so that nodes can have an arbitrary maximum number of elements.

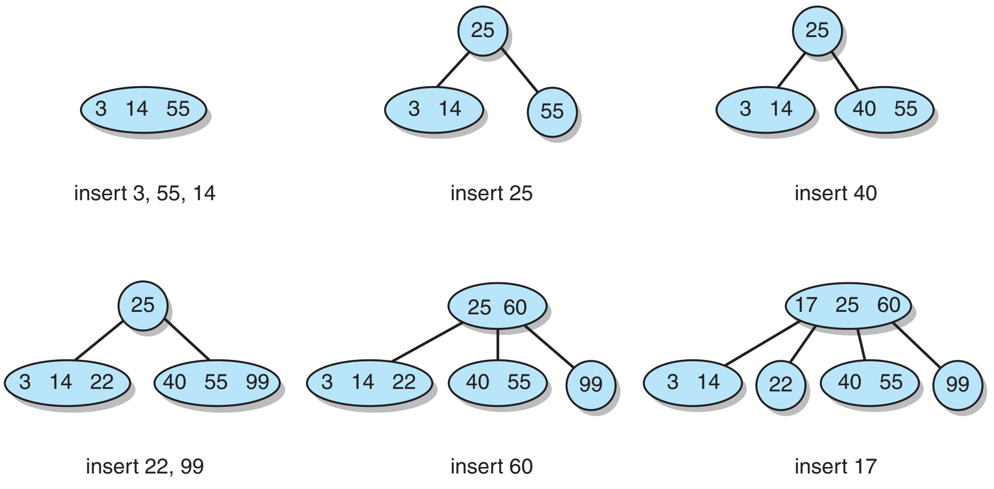


FIGURE 12.12 Insertions into a 2-4 tree

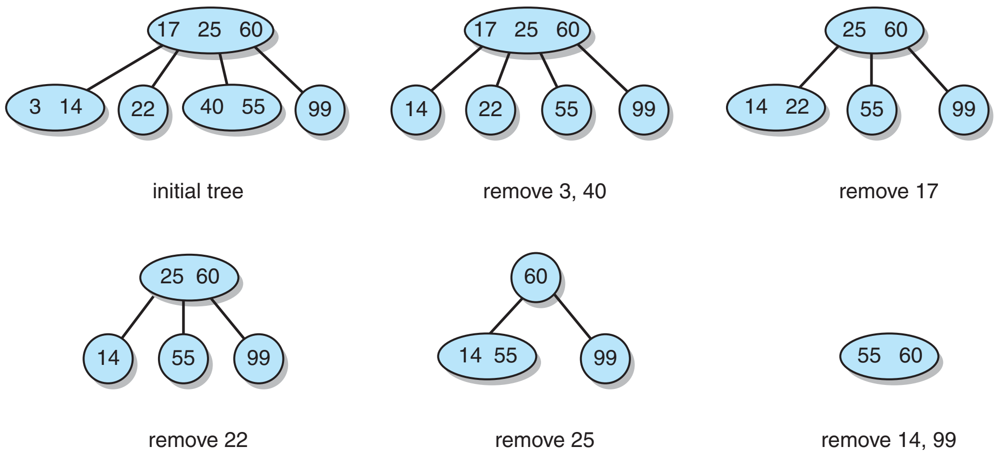


FIGURE 12.13 Removals from a 2-4 tree

B-trees of order m have the following properties:

- The root has at least two subtrees unless it is a leaf.
- Each non-root internal node n holds k–1 elements and k children where $\lceil m/2 \rceil \leq k \leq m$.
- Each leaf n holds k-1 elements where $\lceil m/2 \rceil \leq k \leq m$.
- All leaves are on the same level.

Figure 12.14 illustrates a B-tree of order 6.

KEY CONCEPT

Access to secondary storage is very slow relative to access to primary storage, which is motivation to use structures such as B-trees.

The reasoning behind the creation and use of B-trees is an interesting study of the effects of algorithm and data structure design. To understand this reasoning, we must understand the context of most all of the collections we have discussed thus far. Our assumption has always been that we were dealing with a collection in primary memory. However, what if the data set that we are manipulating is too

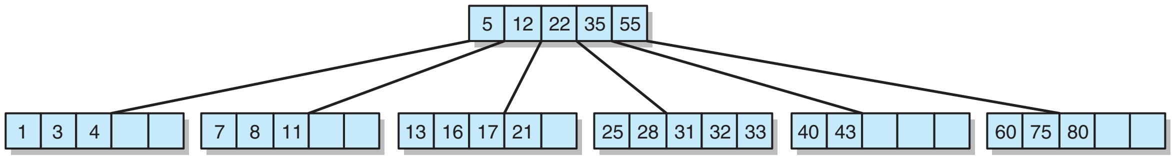


FIGURE 12.14 A B-tree of order 6

large for primary memory? In that case, our data structure would be paged in and out of memory from disk or some other secondary storage device. An interesting thing happens to time complexity once a secondary storage device is involved. No longer is the time to access an element of the collection simply a function of how many comparisons are needed to find the element. Now we must also consider the access time of the secondary storage device and how many separate accesses we will make to that device.

In the case of a disk, this access time consists of seek time (the time it takes to position the read-write head over the appropriate track on the disk), rotational delay (the time it takes to spin the disk to the correct sector), and the transfer time (the time it takes to transfer a block of memory from the disk into primary memory). Adding this "physical" complexity to the access time for a collection can be very costly. Access to secondary storage devices is very slow relative to access to primary storage.

Given this added time complexity, it makes sense to develop a structure that minimizes the number of times the secondary storage device must be accessed. A B-tree can be just such a structure. B-trees are typically tuned so that the size of a node is the same as the size of a block on secondary storage. In this way, we get the maximum amount of data for each disk access. Because B-trees can have many more elements per node than a binary tree, they are much flatter structures than binary trees. This reduces the number of nodes and/or blocks that must be accessed, thus improving performance.

We have already demonstrated the processes of insertion and removal of elements for 2-3 and 2-4 trees, both of which are B-trees. The process for any order m B-tree is similar. Let's now briefly examine some interesting variations of B-trees that were designed to solve specific problems.

B*-trees

One of the potential problems with a B-tree is that although we are attempting to minimize access to secondary storage, we have actually created a data structure that may be half empty. To minimize this problem, B*-trees were developed. *B*-trees* have all of the same properties as B-trees except that, instead of each node having k children where $\lceil m/2 \rceil \leq k \leq m$, in a B*-tree, each node has k children where $\lceil (2m-1)/3 \rceil \leq k \leq m$. This means that each non-root node is at least two-thirds full.

This is accomplished by delaying splitting of nodes by rebalancing across siblings. Once siblings are full, instead of splitting one node into two, creating two half-full nodes, we split two nodes into three, creating three two-thirds full nodes.

B+-trees

Another potential problem with B-trees is sequential access. As with any tree, we can use an inorder traversal to look at the elements of the tree sequentially. However, this means that we are no longer taking advantage of the blocking structure of secondary storage. In fact, we have made it much worse, because now we will access each block containing an internal node many separate times as we pass through it during the traversal.

B+-trees provide a solution to this problem. In a B-tree, each element appears only once in the tree, regardless of whether it appears in an internal node or in a leaf. In a B+-tree, each element appears in a leaf, regardless of whether it appears in an internal node. Elements appearing in an internal node will be listed again as the inorder successor (which is a leaf) of their position in the internal node. Additionally, each leaf node will maintain a pointer to the following leaf node. In this way, a B+-tree provides indexed access through the B-tree structure and sequential access through a linked list of leaves. Figure 12.15 illustrates this strategy.

Analysis of B-trees

With balanced binary search trees, we were able to say that searching for an element in the tree was $O(\log_2 n)$. This is because, at worst, we had to search a single path from the root to a leaf in the tree and, at worst, the length of that path would be $\log_2 n$. Analysis of B-trees is similar. At worst, searching a B-tree, we will have to search a single path from the root to a leaf and, at worst, that path length will be $\log_m n$, where m is the order of the B-tree and n is the number of elements in the tree. However, finding the appropriate node is only part of the search. The other part of the search is finding the appropriate path from each node and then finding the target element in a given node. Because there are up to m–1 elements per node, it may take up to m–1 comparisons per node to find the appropriate path and/or to find the appropriate element. Thus, the analysis of a search of a B-tree yields $O((m-1)\log_m n)$. Because for any given implementation, m is a constant, we can say that searching a B-tree is $O(\log n)$.

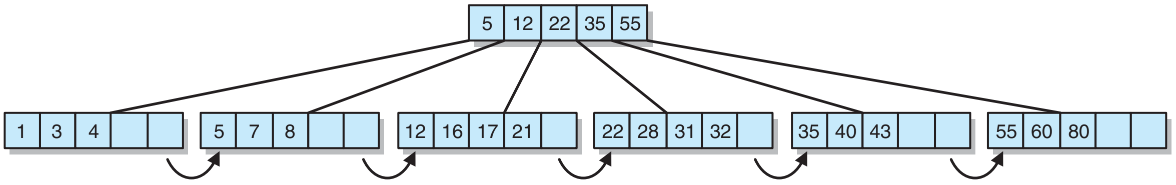


FIGURE 12.15 A B+-tree of order 6

The analysis of insertion into and deletion from a B-tree is similar and is left as an exercise.

12.5 Implementation Strategies for B-Trees

We have already discussed insertion of elements into B-trees, removal of elements from B-trees, and the balancing mechanisms necessary to maintain the properties of a B-tree. What remains is to discuss strategies for storing B-trees. Keep in mind that the B-tree structure was developed specifically to address the issue of a collection that must move in and out of primary memory from secondary storage. If we attempt to use object reference variables to create a linked implementation, we are actually storing a primary memory address for an object. Once that object is moved back to secondary storage, that address is no longer valid. Therefore, if interaction with secondary memory is part of your motivation to use a B-tree, then an array implementation may be a better solution.

KEY CONCEPT

Arrays may provide a better solution both within a B-tree node and for collecting B-tree nodes because they are effective in both primary memory and secondary storage.

A solution is to think of each node as a pair of arrays. The first array would be an array of m–1 elements and the second array would be an array of m children. Next, if we think of the tree itself as one large array of nodes, then the elements stored in the array of children in each node would simply be integer indexes into this array of nodes.

In primary memory, this strategy works because, using an array, as long as we know the index position of the element within the array, it does not matter to us where the array is loaded in primary memory. For secondary memory, this same strategy works because, given that each node is of fixed length, the address in memory of any given node is given by:

The base address of the file + (index of the node – 1) × length of a node.

The array implementations of 2-3, 2-4, and larger B-trees are left as a programming project.

Summary of Key Concepts

- A multi-way search tree can have more than two children per node and can store more than one element in each node.
- A 2-3 tree contains nodes that contain either one or two elements and have zero, two, or three children.
- Inserting an element into a 2-3 tree can have a ripple effect up the tree.
- If the propagation effect of a 2-3 tree insertion causes the root to split, the tree increases in height.
- A 2-4 tree expands on the concept of a 2-3 tree to include the use of 4-nodes.
- A B-tree extends the concept of 2-3 and 2-4 trees so that nodes can have an arbitrary maximum number of elements.
- Access to secondary storage is very slow relative to access to primary storage, which is motivation to use structures such as B-trees.
- Arrays may provide a better solution both within a B-tree node and for collecting B-tree nodes because they are effective in both primary memory and secondary storage.

Self-Review Questions

SR 12.1 Describe the nodes in a 2-3 tree.

SR 12.2 When does a node in a 2-3 tree split?

SR 12.3 How can splitting a node in a 2-3 tree affect the rest of the tree?

SR 12.4 Describe the process of deleting an element from a 2-3 tree.

SR 12.5 Describe the nodes in a 2-4 tree.

SR 12.6 How do insertions and deletions in a 2-4 tree compare to insertions and deletions in a 2-3 tree?

SR 12.7 When is rotation no longer an option for rebalancing a 2-3 tree after a deletion?

Exercises

EX 12.1 Draw the 2-3 tree that results from adding the following elements into an initially empty tree:

34 45 3 87 65 32 1 12 17

EX 12.2 Using the resulting tree from Exercise 12.1, draw the resulting tree after removing each of the following elements:

3 87 12 17 45

EX 12.3 Repeat Exercise 12.1 using a 2-4 tree.

EX 12.4 Repeat Exercise 12.2 using the resulting 2-4 tree from Exercise 12.3.

EX 12.5 Draw the order 8 B-tree that results from adding the following elements into an initially empty tree:

34 45 3 87 65 32 1 12 17 33 55 23 67 15 39 11 19 47

EX 12.6 Draw the B-tree that results from removing the following from the resulting tree from Exercise 12.5:

1 12 17 33 55 23 19 47

EX 12.7 Describe the complexity (order) of insertion into a B-tree.

EX 12.8 Describe the complexity (order) of deletion from a B-tree.

Programming Projects

PP 12.1 Create an implementation of a 2-3 tree using the array strategy discussed in Section 12.5.

PP 12.2 Create an implementation of a 2-3 tree using a linked strategy.

PP 12.3 Create an implementation of a 2-4 tree using the array strategy discussed in Section 12.5.

PP 12.4 Create an implementation of a 2-4 tree using a linked strategy.

PP 12.5 Create an implementation of an order 7 B-tree using the array strategy discussed in Section 12.5.

PP 12.6 Create an implementation of an order 9 B+-tree using the array strategy discussed in Section 12.5.

PP 12.7 Create an implementation of an order 11 B*-tree using the array strategy discussed in Section 12.5.

PP 12.8 Implement a graphical system to manage employees using an employee id, employee name, and years of service. The system should use an order 7 B-tree to store employees, and it must provide the ability to add and remove employees. After each operation, your system must update a sorted list of employees sorted by name on the screen.

Answers to Self-Review Questions

SRA 12.1 A 2-3 tree node can have either one element or two, and can have no children, two children, or three children. If it has one element, then it is a 2-node and has either no children or two children. If it has two elements, then it is a 3-node and has either no children or three children.

SRA 12.2 A 2-3 tree node splits when it has three elements. The smallest element becomes a 2-node, the largest element becomes a 2-node, and the middle element is promoted or propagated to the parent node.

SRA 12.3 If the split and resulting propagation forces the root node to split, then it will increase the height of the tree.

SRA 12.4 Deletion from a 2-3 tree falls into one of three cases. Case 1, deletion of an element from a 3-node leaf, means simply removing the element and has no impact on the rest of the tree. Case 2, deletion of an element from a 2-node leaf, results in one of four cases. Case 2.1, deletion of an element from a 2-node that has a 3-node sibling, is resolved by rotating either the inorder predecessor or inorder successor of the parent, depending upon whether the 3-node is a left child or a right child, around the parent. Case 2.2, deletion of an element from a 2-node when there is a 3-node leaf elsewhere in the tree, is resolved by rotating an element out of that 3-node and propagating that rotation until a sibling of the node being deleted becomes a 3-node; then this case becomes case 2.1. Case 2.3, deletion of a 2-node where there is a 3-node internal node, can be resolved through rotation as well. Case 2.4, deletion of a 2-node when there are no 3-nodes in the tree, is resolved by reducing the height of the tree.

SRA 12.5 Nodes in a 2-4 tree are exactly like those of a 2-3 tree except that 2-4 trees also allow 4-nodes, or nodes containing three elements and having four children.

SRA 12.6 Insertions and deletions in a 2-4 tree are exactly like those of a 2-3 tree except that splits occur when there are four elements instead of three as in a 2-3 tree.

SRA 12.7 If all of the nodes in a 2-3 tree are 2-nodes, then rotation is not an option for rebalancing.

References

Bayer, R. "Symmetric Binary B-trees: Data Structure and Maintenance Algorithms." *Acta Informatica* (1972): 290–306.

Comer, D. "The Ubiquitous B-Tree." *Computing Surveys 11*(1979): 121–137.

Wedeking, H. "On the Selection of Access Paths in a Data Base System." In *Data Base Management*, edited by J. W. Klimbie and K. L. Koffeman, 385–397. Amsterdam: North-Holland, 1974.

13 Graphs

In Chapter 9, we introduced the concept of a tree, a nonlinear structure defined by the concept that each node in the tree, other than the root node, has exactly one parent. If we were to violate that premise and allow each node in the tree to be connected to a variety of other nodes with no notion of parent or child, the result would be the concept of a graph, which we explore in this chapter. Graphs and graph theory make up entire subdisciplines of both mathematics and computer science. In this chapter, we introduce the basic concepts of graphs and their implementation.

CHAPTER OBJECTIVES

- Define undirected graphs
- Define directed graphs
- Define weighted graphs or networks
- Explore common graph algorithms

13.1 Undirected Graphs

Like trees, a graph is made up of nodes and the connections between those nodes. In graph terminology, we refer to the nodes as *vertices* and refer to the connections among them as *edges*. Vertices are typically referenced by a name or a label. For example, we might label vertices A, B, C, and D. Edges are referenced by a pairing of the vertices that they connect. For example, we might have an edge (A, B), which means there is an edge from vertex A to vertex B.

KEY CONCEPT
An undirected graph is a graph where the pairings representing the edges are unordered.

An *undirected graph* is a graph where the pairings representing the edges are unordered. Thus, listing an edge as (A, B) means that there is a connection between A and B that can be traversed in either direction. In an undirected graph, listing an edge as (A, B) means exactly the same thing as listing the edge as (B, A). Figure 13.1 illustrates the following undirected graph:

Vertices:	A, B, C, D
Edges:	(A, B), (A, C), (B, C), (B, D), (C, D)

KEY CONCEPT
Two vertices in a graph are adjacent if there is an edge connecting them.

Two vertices in a graph are *adjacent* if there is an edge connecting them. For example, in the graph of Figure 13.1, vertices A and B are adjacent while vertices A and D are not. Adjacent vertices are sometimes referred to as *neighbors*. An edge of a graph that connects a vertex to itself is called a *self-loop* or a *sling* and is represented by listing the vertex twice. For example, listing an edge (A, A) would mean that there is a sling connecting A to itself.

KEY CONCEPT
An undirected graph is considered complete if it has the maximum number of edges connecting vertices.

An undirected graph is considered *complete* if it has the maximum number of edges connecting vertices. For the first vertex, it requires (n–1) edges to connect it to the other vertices. For the second vertex, it requires only (n–2) edges because it is already connected to the first vertex. For the third vertex, it requires (n–3) edges. This sequence continues until the final vertex requires no additional edges because all the other vertices

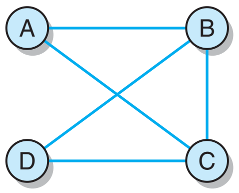


FIGURE 13.1 An example undirected graph

have already been connected to it. Remember from Chapter 2 that the summation from 1 to n is:

$$\sum_{1}^{n} i = n(n + 1)/2$$

Thus, in this case, since we are only summing from 1 to (n–1), the resulting summation is:

$$\sum_{1}^{n-1} i = n(n - 1)/2$$

This means that for any undirected graph with n vertices, it would require n(n–1)/2 edges to make the graph complete. This, of course, assumes that none of those edges are slings.

A *path* is a sequence of edges that connect two vertices in a graph. For example, in our graph from Figure 13.1, A, B, D is a path from A to D. Notice that each sequential pair, (A, B) and then (B, D), is an edge. A path in an undirected graph is bi-directional. For example, A, B, D is the path from A to D, but because the edges are undirected, the inverse, D, B, A, is also the path from D to A. The *length* of a path is the number of edges in the path (or the number of vertices – 1). So for our previous example, the path length is 2. Notice that this definition of path length is identical to the definition that we used in discussing trees. In fact, trees are a special case of graphs.

KEY CONCEPT

A path is a sequence of edges that connects two vertices in a graph.

KEY CONCEPT

A cycle is a path in which the first and last vertices are the same and none of the edges are repeated.

An undirected graph is considered *connected* if for any two vertices in the graph, there is a path between them. Our graph from Figure 13.1 is connected. The same graph with a minor modification is not connected, as illustrated in Figure 13.2.

Vertices:	A, B, C, D
Edges:	(A, B), (A, C), (B, C)

A *cycle* is a path in which the first and last vertices are the same and none of the edges are repeated. In Figure 13.2, we would say that the path A, B, C, A is a cycle. A graph that has no cycles is called *acyclic*. Earlier we mentioned the

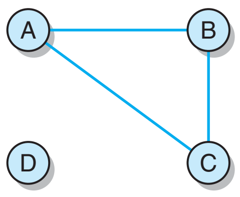


FIGURE 13.2 An example undirected graph that is not connected

KEY CONCEPT
An undirected tree is a connected, acyclic, undirected graph with one element designated as the root.

relationship between graphs and trees. Now that we have introduced these definitions, we can formalize that relationship. An undirected tree is a connected, acyclic, undirected graph with one element designated as the root.

13.2 Directed Graphs

A *directed graph*, sometimes referred to as a *digraph*, is a graph where the edges are ordered pairs of vertices. This means that the edges (A, B) and (B, A) are separate, directional edges in a directed graph. In our previous example, we had the following description for an undirected graph:

Vertices:	A, B, C, D
Edges:	(A, B), (A, C), (B, C), (B, D), (C, D)

Figure 13.3 shows what happens if we interpret this earlier description as a directed graph. We represent each of the edges now with the direction of traversal specified by the ordering of the vertices. For example, the edge (A, B) allows traversal from A to B but not the other direction.

KEY CONCEPT
A directed graph, sometimes referred as a digraph, is a graph where the edges are ordered pairs of vertices.

Our previous definitions change slightly for directed graphs. For example, a path in a directed graph is a sequence of directed edges that connects two vertices in a graph. In our undirected graph, we listed the path A, B, D, as the path from A to D, and that is still true in our directed interpretation of the graph description. However, paths in a directed graph are not bi-directional, so the inverse is no longer true: D, B, A is not a valid path from D to A unless we were to add directional edges (D, B) and (B, A).

KEY CONCEPT
A path in a directed graph is a sequence of directed edges that connects two vertices in a graph.

Our definition for a connected directed graph sounds the same as it did for undirected graphs. A directed graph is connected if for any two vertices in the graph, there is a path between them. However, keep in mind that our definition of path is different. Look at the two graphs shown in Figure 13.4. The first one is connected. The second one, how-

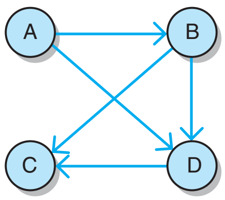


FIGURE 13.3 An example directed graph

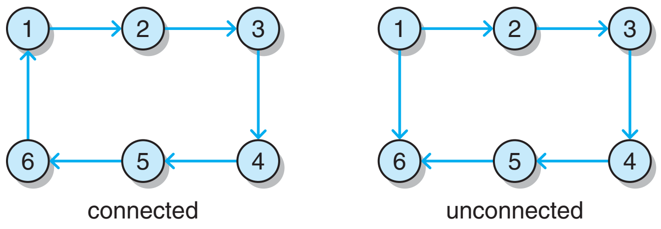


FIGURE 13.4 An example of connected and unconnected directed graphs

ever, is not connected because there is no path from any other vertex to vertex 1.

If a directed graph has no cycles, it is possible to arrange the vertices such that vertex A precedes vertex B if an edge exists from A to B. The order of vertices resulting from this arrangement is called *topological order* and is very useful for examples such as course prerequisites.

As we discussed earlier, trees are graphs. In fact, most of our previous work with trees actually focused on directed trees. A directed tree is a directed graph that has an element designated as the root and has the following properties:

- There are no connections from other vertices to the root.
- Every non-root element has exactly one connection to it.
- There is a path from the root to every other vertex.

13.3 Networks

A *network*, or a *weighted graph*, is a graph with weights or costs associated with each edge. Figure 13.5 shows an undirected network of the connections and the airfares between cities. This weighted graph or network could then be used to determine the cheapest path from one city to another. The weight of a path in a weighted graph is the sum of the weights of the edges in the path.

KEY CONCEPT
A network, or a weighted graph, is a graph with weights or costs associated with each edge.

Networks may be either undirected or directed depending upon the need. Take our airfare example from Figure 13.5. What if the airfare to fly from New York to Boston is one price but the airfare to fly from Boston to New York is a different price? This would be an excellent application of a directed network, as illustrated in Figure 13.6.

For networks, we represent each edge with a triple including the starting vertex, ending vertex, and the weight. Keep in mind, for undirected networks, the starting and ending vertices could be swapped with no impact. However, for directed

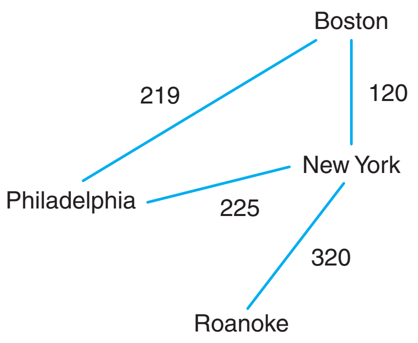


FIGURE 13.5 An undirected network

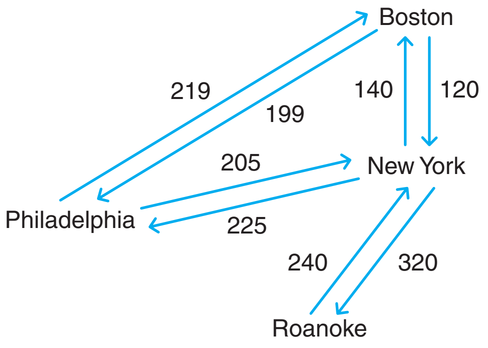


FIGURE 13.6 A directed network

networks, a triple must be included for every directional connection. For example, the network of Figure 13.6 would be represented as follows:

Vertices:	Boston, New York, Philadelphia, Roanoke
Edges:	(Boston, New York, 120), (Boston, Philadelphia, 199), (New York, Boston, 140), (New York, Philadelphia, 225), (New York, Roanoke, 320), (Philadelphia, Boston, 219), (Philadelphia, New York, 205), (Roanoke, New York, 240)

13.4 Common Graph Algorithms

There are a number of common graph algorithms that may apply to undirected graphs, directed graphs, and/or networks. These include various traversal algorithms similar to what we explored with trees, as well as algorithms for finding the shortest path, algorithms for finding the least costly path in a network, and algorithms to answer simple questions about the graph such as whether or not the graph is connected or what the shortest path is between two vertices.

Traversals

In our discussion of trees in Chapter 9, we defined four types of traversals and then implemented them as iterators: preorder traversal, inorder traversal, postorder traversal, and level-order traversal. Because we know that a tree is a graph, we know that for certain types of graphs these traversals would still apply. Generally, however, we divide graph traversal into two categories: a *breadth-first* traversal, which behaves very much like the level-order traversal of a tree, and a *depth-first* traversal, which behaves very much like the preorder traversal of a tree. One difference here is that there is not a root node. Thus our traversal may start at any vertex in the graph.

We can construct a breadth-first traversal for a graph using a queue and an unordered list. We will use the queue (traversal-queue) to manage the traversal and the unordered list (result-list) to build our result. The first step is to enqueue the starting vertex into the traversal-queue and mark the starting vertex as visited. We then begin a loop that will continue until the traversal-queue is empty. Within this loop, we will take the first vertex off of the traversal-queue and add that vertex to the rear of the result-list. Next, we will enqueue each of the vertices that are adjacent to the current one, and have not already been marked as visited, into the traversal-queue, mark each of them as visited, and then repeat the loop. We simply repeat this process for each of the visited vertices until the traversal-queue is empty, meaning we can no longer reach any new vertices. The result-list now contains the vertices in breadth-first order from the given starting point. Very similar logic can be used to construct a breadth-first iterator. The `iteratorBFS` shows an iterative algorithm for this traversal for an array implementation of a graph. The determination of vertices that are adjacent to the current one depends upon the implementation we choose to represent edges in a graph. This particular method assumes an implementation using an adjacency matrix. We will discuss this further in Section 13.5.

A depth-first traversal for a graph can be constructed using virtually the same logic by simply replacing the traversal-queue with a traversal-stack. One other difference in the algorithm, however, is that we do not want to mark a vertex as visited until it has been added to the result-list. The `iteratorDFS` method illustrates this algorithm for an array implementation of a graph.

KEY CONCEPT
The only difference between a depth-first traversal of a graph and a breadth-first traversal is the use of a stack instead of a queue to manage the traversal.

```java
/**
 * Returns an iterator that performs a breadth-first search
 * traversal starting at the given index.
 *
 * @param startIndex  the index to begin the search from
 * @return            an iterator that performs a breadth-first traversal
 */
public Iterator<T> iteratorBFS(int startIndex)
{
   Integer x;
   LinkedQueue<Integer> traversalQueue = new LinkedQueue<Integer>();
   ArrayUnorderedList<T> resultList = new ArrayUnorderedList<T>();

   if (!indexIsValid(startIndex))
      return resultList.iterator();

   boolean[] visited = new boolean[numVertices];
   for (int i = 0; i < numVertices; i++)
      visited[i] = false;

   traversalQueue.enqueue(new Integer(startIndex));
   visited[startIndex] = true;

   while (!traversalQueue.isEmpty())
   {
      x = traversalQueue.dequeue();
      resultList.addToRear(vertices[x.intValue()]);

      /** Find all vertices adjacent to x that have not been visited
          and queue them up */

      for (int i = 0; i < numVertices; i++)
      {
         if (adjMatrix[x.intValue()][i] && !visited[i])
         {
            traversalQueue.enqueue(new Integer(i));
            visited[i] = true;
         }
      }
   }
   return resultList.iterator();
}
```

```java
/**
 * Returns an iterator that performs a depth-first search
 * traversal starting at the given index.
 *
 * @param startIndex  the index to begin the search traversal from
 * @return            an iterator that performs a depth-first traversal
 */
public Iterator<T> iteratorDFS(int startIndex)
{
   Integer x;
   boolean found;
   LinkedStack<Integer> traversalStack = new LinkedStack<Integer>();
   ArrayUnorderedList<T> resultList = new ArrayUnorderedList<T>();
   boolean[] visited = new boolean[numVertices];

   if (!indexIsValid(startIndex))
      return resultList.iterator();

   for (int i = 0; i < numVertices; i++)
      visited[i] = false;

   traversalStack.push(new Integer(startIndex));
   resultList.addToRear(vertices[startIndex]);
   visited[startIndex] = true;

   while (!traversalStack.isEmpty())
   {
      x = traversalStack.peek();
      found = false;

      /** Find a vertex adjacent to x that has not been visited
          and push it on the stack */
      for (int i = 0; (i < numVertices) && !found; i++)
      {
         if (adjMatrix[x.intValue()][i] && !visited[i])
         {
            traversalStack.push(new Integer(i));
            resultList.addToRear(vertices[i]);
            visited[i] = true;
            found = true;
         }
      }
      if (!found && !traversalStack.isEmpty())
         traversalStack.pop();
   }
   return resultList.iterator();
}
```

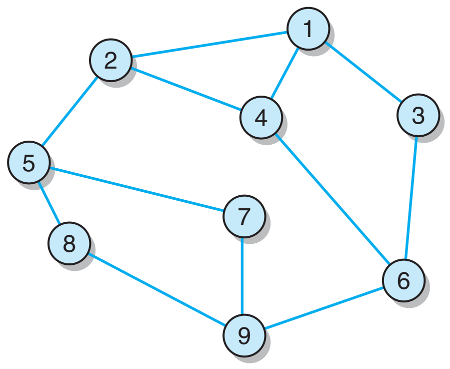


FIGURE 13.7 A traversal example

Let's look at an example. Figure 13.7 shows a sample undirected graph where each vertex is labeled with an integer. For a breadth-first traversal starting from vertex 9, we do the following:

1. Add 9 to the traversal-queue and mark it as visited.
2. Dequeue 9 from the traversal-queue.
3. Add 9 on the result-list.
4. Add 6, 7, and 8 to the traversal-queue, marking each of them as visited.
5. Dequeue 6 from the traversal-queue.
6. Add 6 on the result-list.
7. Add 3 and 4 to the traversal-queue, marking them both as visited.
8. Dequeue 7 from the traversal-queue and add it to the result-list.
9. Add 5 to the traversal-queue, marking it as visited.
10. Dequeue 8 from the traversal-queue and add it to the result-list. (We do not add any new vertices to the traversal-queue because there are no neighbors of 8 that have not already been visited.)
11. Dequeue 3 from the traversal-queue and add it to the result-list.
12. Add 1 to the traversal-queue, marking it as visited.
13. Dequeue 4 from the traversal-queue and add it to the result-list.
14. Add 2 to the traversal-queue, marking it as visited.
15. Dequeue 5 from the traversal-queue and add it to the result-list. (Because there are no unvisited neighbors, we continue without adding anything to the traversal-queue.)
16. Dequeue 1 from the traversal-queue and add it to the result-list. (Because there are no unvisited neighbors, we continue without adding anything to the traversal-queue.)
17. Dequeue 2 from the traversal-queue and add it to the result-list.

Thus, the result-list now contains the breadth-first order starting at vertex 9: 9, 6, 7, 8, 3, 4, 5, 1, and 2. Try tracing a depth-first search on the same graph from Figure 13.7.

Of course, both of these algorithms could be expressed recursively. For example, the following algorithm recursively defines a depth-first search:

```
DepthFirstSearch(node x)
{
   visit(x)
   result-list.addToRear(x)
   for each node y adjacent to x
         if y not visited
               DepthFirstSearch(y)
}
```

Testing for Connectivity

In our earlier discussion, we defined a graph as *connected* if for any two vertices in the graph, there is a path between them. This definition holds true for both undirected and directed graphs. Given the algorithm we just discussed, there is a simple solution to the question of whether or not a graph is connected: The graph is connected if and only if for each vertex v in a graph containing n vertices, the size of the result of a breadth-first traversal starting at v is n.

KEY CONCEPT

A graph is connected if and only if the number of vertices in the breadth-first traversal is the same as the number of vertices in the graph regardless of the starting vertex.

Let's look at the example undirected graphs in Figure 13.8. We stated earlier that the graph on the left is connected and that the graph on the right is not. Let's confirm that by following our algorithm. Figure 13.9 shows the breadth-first traversals for the graph on the left using each of the vertices as a starting point. As you can see, all of the traversals yield n = 4 vertices, thus the graph is connected. Figure 13.10 shows the breadth-first traversals for the graph on the right using each of the vertices as a starting point. Notice that not only do none of the traversals contain n = 4 vertices, but the one starting at vertex D has only the one vertex. Thus the graph is not connected.

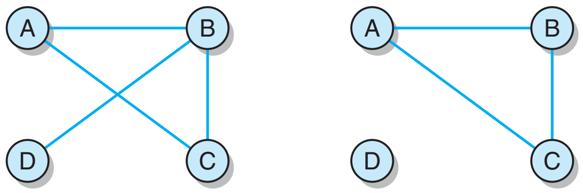


FIGURE 13.8 Connectivity in an undirected graph

Starting Vertex	Breadth-First Traversal
A	A, B, C, D
B	B, A, D, C
C	C, B, A, D
D	D, B, A, C

FIGURE 13.9 Breadth-first traversal for a connected undirected graph

Starting Vertex	Breadth-First Traversal
A	A, B, C
B	B, A, C
C	C, B, A
D	D

FIGURE 13.10 Breadth-first traversal for an unconnected undirected graph

Minimum Spanning Trees

A *spanning tree* is a tree that includes all of the vertices of a graph and some, but possibly not all, of the edges. Because trees are also graphs, for some graphs, the graph itself will be a spanning tree, and thus the only spanning tree for that graph will include all of the edges. Figure 13.11 shows a spanning tree for our graph from Figure 13.7.

KEY CONCEPT
A spanning tree is a tree that includes all of the vertices of a graph and some, but possibly not all, of the edges.

KEY CONCEPT
A minimum spanning tree is a spanning tree where the sum of the weights of the edges is less than or equal to the sum of the weights for any other spanning tree for the same graph.

One interesting application of spanning trees is to find a minimum spanning tree for a weighted graph. A *minimum spanning tree* is a spanning tree where the sum of the weights of the edges is less than or equal to the sum of the weights for any other spanning tree for the same graph.

The algorithm for developing a minimum spanning tree was developed by Prim (1957) and is quite elegant. As we discussed earlier, each edge is represented by a triple including the starting vertex, ending vertex, and the weight. We then pick an arbitrary starting vertex (it does not matter which one) and add it to our minimum spanning tree (MST). Next we add all of the edges that

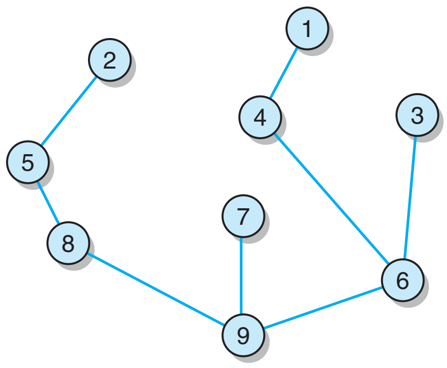


FIGURE 13.11 A spanning tree

include our starting vertex to a minheap ordered by weight. Keep in mind that if we are dealing with a directed network, we will only add edges that start at the given vertex.

Next we remove the minimum edge from the minheap and add the edge and the new vertex to our MST. Next we add to our minheap all of the edges that include this new vertex and whose other vertex is not already in our MST. We continue this process until either our MST includes all of the vertices in our original graph or the minheap is empty. Figure 13.12 shows a weighted network and its associated minimum spanning tree. The `getMST` method illustrates this algorithm.

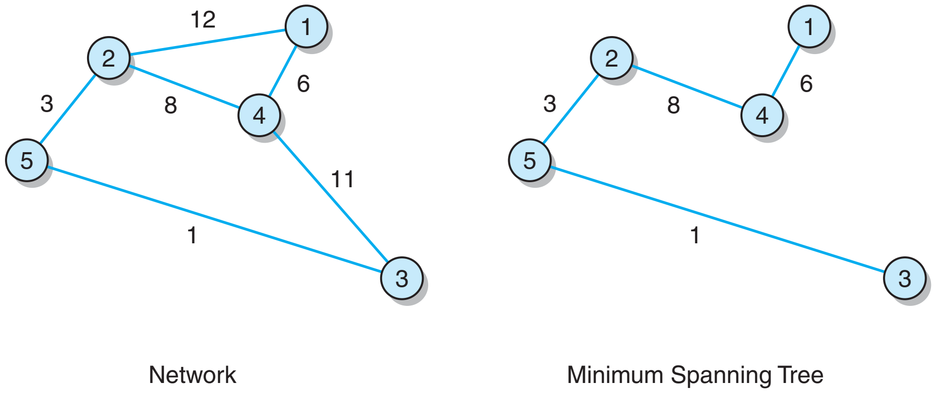


FIGURE 13.12 A minimum spanning tree

```
/**
 * Returns a minimum spanning tree of the network.
 *
 * @return a minimum spanning tree of the network
 */
public Network mstNetwork()
{
   int x, y;
   int index;
   double weight;
   int[] edge = new int[2];
   Heap<Double> minHeap = new Heap<Double>();
   Network<T> resultGraph = new Network<T>();

   if (isEmpty() || !isConnected())
      return resultGraph;

   resultGraph.adjMatrix = new double[numVertices][numVertices];
   for (int i = 0; i < numVertices; i++)
      for (int j = 0; j < numVertices; j++)
         resultGraph.adjMatrix[i][j] = Double.POSITIVE_INFINITY;
   resultGraph.vertices = (T[])(new Object[numVertices]);

   boolean[] visited = new boolean[numVertices];
   for (int i = 0; i < numVertices; i++)
      visited[i] = false;

   edge[0] = 0;
   resultGraph.vertices[0] = this.vertices[0];
   resultGraph.numVertices++;
   visited[0] = true;

   /** Add all edges, which are adjacent to the starting vertex,
       to the heap */

   for (int i = 0; i < numVertices; i++)
         minHeap.addElement(new Double(adjMatrix[0][i]));

   while ((resultGraph.size() < this.size()) && !minHeap.isEmpty())
   {

      /** Get the edge with the smallest weight that has exactly
          one vertex already in the resultGraph */
      do
      {
            weight = (minHeap.removeMin()).doubleValue();
            edge = getEdgeWithWeightOf(weight, visited);
         } while (!indexIsValid(edge[0]) || !indexIsValid(edge[1]));

         x = edge[0];
         y = edge[1];
         if (!visited[x])
            index = x;
         else
            index = y;

         /** Add the new edge and vertex to the resultGraph */
         resultGraph.vertices[index] = this.vertices[index];
         visited[index] = true;
         resultGraph.numVertices++;

         resultGraph.adjMatrix[x][y] = this.adjMatrix[x][y];
         resultGraph.adjMatrix[y][x] = this.adjMatrix[y][x];

         /** Add all edges, that are adjacent to the newly added vertex,
             to the heap */
         for (int i = 0; i < numVertices; i++)
         {
            if (!visited[i] && (this.adjMatrix[i][index] <
                                Double.POSITIVE_INFINITY))
            {
               edge[0] = index;
               edge[1] = i;
               minHeap.addElement(new Double(adjMatrix[index][i]));
            }
         }
      }
      return resultGraph;
   }
```

Determining the Shortest Path

There are two possibilities for determining the "shortest" path in a graph. The first, and perhaps simplest, possibility is to determine the literal shortest path between a starting vertex and a target vertex, meaning the least number of edges between the two vertices. This turns out to be a simple variation of our earlier breadth-first traversal algorithm.

To convert this algorithm to find the shortest path, we simply store two additional pieces of information for each vertex during our traversal: the path length from the starting vertex to this vertex, and the vertex that is the predecessor of this vertex in that path. Then we modify our loop to terminate when we reach our target vertex. The path length for the shortest path is simply the path length to the predecessor of the target + 1, and if we wish to output the vertices along the shortest path, we can simply backtrack along the chain of predecessors.

The second possibility for determining the shortest path is to look for the cheapest path in a weighted graph. Dijkstra (1959) developed an algorithm for this possibility that is similar to our previous algorithm. However, instead of using a queue of vertices that causes us to progress through the graph in the order we encounter vertices, we use a minheap or a priority queue storing vertex and weight pairs based upon total weight (the sum of the weights from the starting vertex to this vertex) so that we always traverse through the graph following the cheapest path first. For each vertex, we must store the label of the vertex, the weight of the cheapest path (thus far) to that vertex from our starting point, and the predecessor of that vertex along that path. On the minheap, we will store vertex and weight pairs for each possible path that we have encountered but not yet traversed. As we remove a vertex, weight pair from the minheap, if we encounter a vertex with a weight less than the one already stored with the vertex, we update the cost.

13.5 Strategies for Implementing Graphs

Let us begin our discussion of implementation strategies by examining what operations would need to be available for a graph. Of course, we would need to be able to add and remove vertices, and add and remove edges from the graph. There will need to be traversals (perhaps breadth first and depth first) beginning with a particular vertex, and these might be implemented as iterators, as we did for binary trees. Other operations like `size`, `isEmpty`, `toString`, and `find` will be useful as well. In addition to these, operations to determine the shortest path from a particular vertex to a particular target vertex, to determine the adjacency of two vertices, to construct a minimum spanning tree, and to test for connectivity would all likely need to be implemented.

Whatever storage mechanism we use for vertices must allow us to mark vertices as visited during traversals and other algorithms. This can be accomplished by simply adding a Boolean variable to the class representing the vertices.

Adjacency Lists

Because trees are graphs, perhaps the best introduction to how we might implement graphs is to consider the discussions and examples that we have already seen concerning the implementation of trees. One might immediately think of using a set of nodes where each node contains an element and n–1 links to other nodes. When we used this strategy with trees, the number of connections from any given node was limited by the order of the tree (e.g., a maximum of two directed edges starting at any particular node in a binary tree). Because of this limitation, we were able to specify, for example, that a binary node had a left and a right child pointer. Even if the binary node was a leaf, the pointer still existed. It was simply set to null.

In the case of a *graph node*, because each node could have up to n–1 edges connecting it to other nodes, it would be better to use a dynamic structure such as a linked list to store the edges within each node. This list is called an *adjacency list*. In the case of a network or weighted graph, each edge would be stored as a triple including the weight. In the case of an undirected graph, an edge (A, B) would appear in the adjacency list of both vertex A and vertex B.

Adjacency Matrices

Keep in mind that we must somehow efficiently (both in terms of space and access time) store both vertices and edges. Because vertices are just elements, we can use any of our collections to store the vertices. In fact, we often talk about a "set of vertices," the term *set* implying an implementation strategy. However, another solution for storing edges is motivated by our use of array implementations of trees, but instead of using a one-dimensional array, we will use a two-dimensional array that we call an *adjacency matrix*. In an adjacency matrix, each position of the two-dimensional array represents an intersection between two vertices in the graph. Each of these intersections is represented by a Boolean value indicating whether or not the two vertices are connected. Figure 13.13 shows the undirected graph that we began with at the beginning of this chapter. Figure 13.14 shows the adjacency matrix for this graph.

For any position (row, column) in the matrix, that position is true if and only if the edge (v_{row}, v_{column}) is in the graph. Because edges in an undirected graph are bi-directional, if (A, B) is an edge in the graph, then (B, A) is also in the graph.

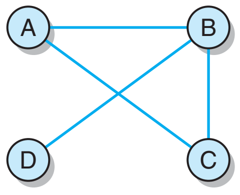


FIGURE 13.13 An undirected graph

	A	B	C	D
A	F	T	T	F
B	T	F	T	T
C	T	T	F	F
D	F	T	F	F

FIGURE 13.14 An adjacency matrix for an undirected graph

Notice that this matrix is symmetrical—that is, each side of the diagonal is a mirror image of the other. The reason for this is that we are representing an undirected graph. For undirected graphs, it may not be necessary to represent the entire matrix but simply one side or the other of the diagonal.

However, for directed graphs, because all of the edges are directional, the result can be quite different. Figure 13.15 shows a directed graph, and Figure 13.16 shows the adjacency matrix for this graph.

Adjacency matrices may also be used with networks or weighted graphs by simply storing an object at each position of the matrix to represent the weight of the edge. Positions in the matrix where edges do not exist would simply be set to null.

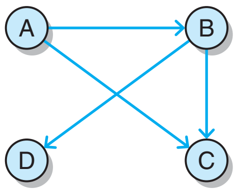


FIGURE 13.15 A directed graph

	A	B	C	D
A	F	T	T	F
B	F	F	T	T
C	F	F	F	F
D	F	F	F	F

FIGURE 13.16 An adjacency matrix for a directed graph

13.6 Implementing Undirected Graphs with an Adjacency Matrix

Like the other collections we have discussed, the first step in implementing a graph is to determine its interface. Listing 13.1 illustrates the `GraphADT` interface. Listing 13.2 illustrates the `NetworkADT` interface that extends the `GraphADT` interface. Note that our interfaces include methods to add and remove vertices, add and remove edges, iterators for both breadth-first and depth-first traversals, methods to determine the shortest path between two vertices and to determine if the graph is connected, and our usual collection of methods to determine the size of the collection, determine if it is empty, and return a string representation of it.

LISTING 13.1

```
/**
 * GraphADT defines the interface to a graph data structure.
 *
 * @author Dr. Chase
 * @author Dr. Lewis
 * @version 1.0, 9/17/2008
 */

package jss2;
import java.util.Iterator;

public interface GraphADT<T>
{
   /**
    * Adds a vertex to this graph, associating object with vertex.
    *
    * @param vertex  the vertex to be added to this graph
    */
   public void addVertex (T vertex);

   /**
    * Removes a single vertex with the given value from this graph.
    *
    * @param vertex  the vertex to be removed from this graph
    */
   public void removeVertex (T vertex);

   /**
    * Inserts an edge between two vertices of this graph.
```

LISTING 13.1 *continued*

```
     *
     * @param vertex1  the first vertex
     * @param vertex2  the second vertex
     */
    public void addEdge (T vertex1, T vertex2);

    /**
     * Removes an edge between two vertices of this graph.
     *
     * @param vertex1  the first vertex
     * @param vertex2  the second vertex
     */
    public void removeEdge (T vertex1, T vertex2);

    /**
     * Returns a breadth first iterator starting with the given vertex.
     *
     * @param startVertex  the starting vertex
     * @return             a breadth first iterator beginning at the given
     *                     vertex
     */
    public Iterator iteratorBFS(T startVertex);

    /**
     * Returns a depth first iterator starting with the given vertex.
     *
     * @param startVertex  the starting vertex
     * @return             a depth first iterator starting at the given vertex
     */
    public Iterator iteratorDFS(T startVertex);

    /**
     * Returns an iterator that contains the shortest path between
     * the two vertices.
     *
     * @param startVertex   the starting vertex
     * @param targetVertex  the ending vertex
     * @return              an iterator that contains the shortest path
     *                      between the two vertices
     */
    public Iterator iteratorShortestPath(T startVertex, T targetVertex);
```

LISTING 13.1 *continued*

```
   /**
    * Returns true if this graph is empty, false otherwise.
    *
    * @return  true if this graph is empty
    */
   public boolean isEmpty();

   /**
    * Returns true if this graph is connected, false otherwise.
    *
    * @return  true if this graph is connected
    */
   public boolean isConnected();

   /**
    * Returns the number of vertices in this graph.
    *
    * @return  the integer number of vertices in this graph
    */
   public int size();

   /**
    * Returns a string representation of the adjacency matrix.
    *
    * @return  a string representation of the adjacency matrix
    */
   public String toString();
}
```

LISTING 13.2

```
/**
 * NetworkADT defines the interface to a network.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 9/17/2008
 */
```

LISTING 13.2 *continued*

```
package jss2;
import java.util.Iterator;

public interface NetworkADT<T> extends GraphADT<T>
{
   /**
    * Inserts an edge between two vertices of this graph.
    *
    * @param vertex1  the first vertex
    * @param vertex2  the second vertex
    * @param weight   the weight
    */
   public void addEdge (T vertex1, T vertex2, double weight);

   /**
    * Returns the weight of the shortest path in this network.
    *
    * @param vertex1  the first vertex
    * @param vertex2  the second vertex
    * @return         the weight of the shortest path in this network
    */
   public double shortestPathWeight(T vertex1, T vertex2);
}
```

Of course, this interface could be implemented a variety of ways, but we will focus our discussion on an adjacency matrix implementation. The other implementations of undirected graphs and networks as well as the implementations of directed graphs and networks are left as programming projects. The header and instance data for our implementation are presented to provide context. Note that the adjacency matrix is represented by a two-dimensional Boolean array.

```
/**
 * Graph represents an adjacency matrix implementation of a graph.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 9/16/2008
 */
```

```
package jss2;
import jss2.exceptions.*;
import java.util.*;

public class Graph<T> implements GraphADT<T>
{
   protected final int DEFAULT_CAPACITY = 10;
   protected int numVertices;  // number of vertices in the graph
   protected boolean[][] adjMatrix;  // adjacency matrix
   protected T[] vertices;  // values of vertices
```

Our constructor simply initializes the number of vertices to zero, constructs the adjacency matrix, and sets up an array of generic objects (`T[]`) to represent the vertices.

```
/**
 * Creates an empty graph.
 */
public Graph()
{
   numVertices = 0;
   this.adjMatrix = new boolean[DEFAULT_CAPACITY][DEFAULT_CAPACITY];
   this.vertices = (T[])(new Object[DEFAULT_CAPACITY]);
}
```

The `addEdge` Method

Once we have established our list of vertices and our adjacency matrix, adding an edge is simply a matter of setting the appropriate locations in the adjacency matrix to true. Our `addEdge` method uses the `getIndex` method to locate the proper indices and calls a different version of the `addEdge` method to make the assignments if the indices are valid.

```
/**
 * Inserts an edge between two vertices of the graph.
 *
 * @param vertex1  the first vertex
 * @param vertex2  the second vertex
 */
```

```
public void addEdge (T vertex1, T vertex2)
{
   addEdge (getIndex(vertex1), getIndex(vertex2));
}
```

```
/**
 * Inserts an edge between two vertices of the graph.
 *
 * @param index1  the first index
 * @param index2  the second index
 */
public void addEdge (int index1, int index2)
{
   if (indexIsValid(index1) && indexIsValid(index2))
   {
      adjMatrix[index1][index2] = true;
      adjMatrix[index2][index1] = true;
   }
}
```

The `addVertex` Method

Adding a vertex to the graph involves adding the vertex in the next available position in the array and setting all of the appropriate locations in the adjacency matrix to false.

```
/**
 * Adds a vertex to the graph, expanding the capacity of the graph
 * if necessary.  It also associates an object with the vertex.
 *
 * @param vertex the vertex to add to the graph
 */
public void addVertex (T vertex)
{
   if (numVertices == vertices.length)
      expandCapacity();

   vertices[numVertices] = vertex;
   for (int i = 0; i <= numVertices; i++)
```

```
      {
         adjMatrix[numVertices][i] = false;
         adjMatrix[i][numVertices] = false;
      }
      numVertices++;
   }
```

The expandCapacity Method

The `expandCapacity` method for our adjacency matrix implementation of a graph is more interesting than the similar method in our other array implementations. It is no longer just a case of expanding one array and copying the contents. Keep in mind that for our graph, we must not only expand the array of vertices and copy the existing vertices into the new array; we must also expand the capacity of the adjacency list and copy the old contents into the new list.

```
   /**
    * Creates new arrays to store the contents of the graph with
    * twice the capacity.
    */
   protected void expandCapacity()
   {
      T[] largerVertices = (T[])(new Object[vertices.length*2]);
      boolean[][] largerAdjMatrix =
            new boolean[vertices.length*2][vertices.length*2];

      for (int i = 0; i < numVertices; i++)
      {
         for (int j = 0; j < numVertices; j++)
         {
            largerAdjMatrix[i][j] = adjMatrix[i][j];
         }
         largerVertices[i] = vertices[i];
      }

      vertices = largerVertices;
      adjMatrix = largerAdjMatrix;
   }
```

Other Methods

The remaining methods for our graph implementation are left as programming projects.

Summary of Key Concepts

- An undirected graph is a graph where the pairings representing the edges are unordered.
- Two vertices in a graph are adjacent if there is an edge connecting them.
- An undirected graph is considered complete if it has the maximum number of edges connecting vertices.
- A path is a sequence of edges that connects two vertices in a graph.
- A cycle is a path in which the first and last vertices are the same and none of the edges are repeated.
- An undirected tree is a connected, acyclic, undirected graph with one element designated as the root.
- A directed graph, sometimes referred as a digraph, is a graph where the edges are ordered pairs of vertices.
- A path in a directed graph is a sequence of directed edges that connects two vertices in a graph.
- A network, or a weighted graph, is a graph with weights or costs associated with each edge.
- The only difference between a depth-first traversal of a graph and a breadth-first traversal is the use of a stack instead of a queue to manage the traversal.
- A graph is connected if and only if the number of vertices in the breadth-first traversal is the same as the number of vertices in the graph regardless of the starting vertex.
- A spanning tree is a tree that includes all of the vertices of a graph and some, but possibly not all, of the edges.
- A minimum spanning tree is a spanning tree where the sum of the weights of the edges is less than or equal to the sum of the weights for any other spanning tree for the same graph.

Self-Review Questions

SR 13.1 What is the difference between a graph and a tree?

SR 13.2 What is an undirected graph?

SR 13.3 What is a directed graph?

SR 13.4 What does it mean to say that a graph is complete?

SR 13.5 What is the maximum number of edges for an undirected graph and the maximum number of edges for a directed graph?

SR 13.6 What is the definition of path and the definition of a cycle?

SR 13.7 What is the difference between a network and a graph?

SR 13.8 What is a spanning tree? A minimum spanning tree?

Exercises

EX 13.1 Draw the undirected graph that is represented by the following:

```
vertices: 1, 2, 3, 4, 5, 6, 7
edges: (1, 2), (1, 4), (2, 3), (2, 4), (3, 7), (4, 7),
(4, 6), (5, 6), (5, 7), (6, 7)
```

EX 13.2 Is the graph from Exercise 13.1 connected? Is it complete?

EX 13.3 List all of the cycles in the graph from Exercise 13.1.

EX 13.4 Draw a spanning tree for the graph of Exercise 13.1.

EX 13.5 Using the same data from Exercise 13.1, draw the resulting directed graph.

EX 13.6 Is the directed graph of Exercise 13.5 connected? Is it complete?

EX 13.7 List all of the cycles in the graph of Exercise 13.5.

EX 13.8 Draw a spanning tree for the graph of Exercise 13.5.

EX 13.9 Consider the weighted graph shown in Figure 13.10. List all of the possible paths from vertex 2 to vertex 3 along with the total weight of each path.

Programming Projects

PP 13.1 Implement an undirected graph using an adjacency list. Keep in mind that you must store both vertices and edges. Your implementation must implement the `GraphADT` interface.

PP 13.2 Repeat Programming Project 13.1 for a directed graph.

PP 13.3 Complete the implementation of a graph using an adjacency matrix that was presented in this chapter.

PP 13.4 Extend the adjacency matrix implementation presented in this chapter to create an implementation of a weighted graph or network.

PP 13.5 Extend the adjacency matrix implementation presented in this chapter to create a directed graph.

PP 13.6 Extend your implementation from Programming Project 13.1 to create a weighted, undirected graph.

PP 13.7 Create a limited airline scheduling system that will allow a user to enter city to city connections and their prices. Your system should then allow a user to enter two cities and should return the shortest path and the cheapest path between the two cities. Your system should report if there is no connection between two cities. Assume an undirected network.

PP 13.8 Repeat Programming Project 13.7 assuming a directed network.

PP 13.9 Create a simple graphical application that will produce a textual representation of the shortest path and the cheapest path between two vertices in a network.

PP 13.10 Create a network routing system that, given the point-to-point connections in the network and the costs of utilizing each, will produce cheapest-path connections from each point to each point in the network, pointing out any disconnected locations.

Answers to Self-Review Questions

SRA 13.1 A graph is the more general concept without the restriction that each node have one and only one parent except for the root, which does not have a parent. In the case of a graph, there is no root, and each vertex can be connected to up to n–1 other vertices.

SRA 13.2 An undirected graph is a graph where the pairings representing the edges are unordered.

SRA 13.3 A directed graph, sometimes referred as a digraph, is a graph where the edges are ordered pairs of vertices.

SRA 13.4 A graph is considered complete if it has the maximum number of edges connecting vertices.

SRA 13.5 The maximum number of edges for an undirected graph is n(n–1)/2. For a directed graph, it is n(n–1).

SRA 13.6 A path is a sequence of edges that connects two vertices in a graph. A cycle is a path in which the first and last vertices are the same and none of the edges are repeated.

SRA 13.7 A network is a graph, either directed or undirected, with weights or costs associated with each edge.

SRA 13.8 A spanning tree is a tree that includes all of the vertices of a graph and some, but possibly not all, of the edges. A minimum spanning tree is a spanning tree where the sum of the weights of the edges is less than or equal to the sum of the weights for any other spanning tree for the same graph.

References

Collins, W. J. *Data Structures: An Object-Oriented Approach*. Reading, Mass.: Addison-Wesley, 1992.

Dijkstra, E. W. "A Note on Two Problems in Connection with Graphs." *Numerische Mathematik 1* (1959): 269–271.

Drosdek, A. *Data Structures and Algorithms in Java*. Pacific Grove, Cal.: Brooks/Cole, 2001.

Prim, R. C. "Shortest Connection Networks and Some Generalizations." *Bell System Technical Journal 36* (1957): 1389–1401.

This page intentionally left blank

14 Hashing

In Chapter 10, we discussed the idea that a binary search tree is, in effect, an efficient implementation of a set or a map. In this chapter, we examine hashing, an approach to implementing a set or map collection that can be even more efficient than binary search trees.

CHAPTER OBJECTIVES

- Define hashing
- Examine various hashing functions
- Examine the problem of collisions in hash tables
- Explore the Java Collections API implementations of hashing

14.1 A Hashing

In all of our discussions of the implementations of collections, we have proceeded with one of two assumptions about the order of elements in a collection:

- Order is determined by the order in which elements are added to and/or removed from our collection, as in the case of stacks, queues, unordered lists, and indexed lists.
- Order is determined by comparing the values of the elements (or some key component of the elements) to be stored in the collection, as in the case of ordered lists and binary search trees.

In this chapter, we will explore the concept of *hashing*, which means that the order—and, more specifically, the location of an item within the collection—is determined by some function of the value of the element to be stored, or some function of a key value of the element to be stored. In hashing, elements are stored in a *hash table*, with their location in the table determined by a *hashing function*. Each location in the table may be referred to as a *cell* or a *bucket*. We will discuss hashing functions further in Section 14.2. We will discuss implementation strategies and algorithms, and we will leave the implementations as programming projects.

KEY CONCEPT
In hashing, elements are stored in a hash table, with their location in the table determined by a hashing function.

Consider a simple example where we create an array that will hold 26 elements. Wishing to store names in our array, we create a hashing function that equates each name to the position in the array associated with the first letter of the name (e.g., a first letter of A would be mapped to position 0 of the array, a first letter of D would be mapped to position 3 of the array, and so on). Figure 14.1 illustrates this scenario after several names have been added.

Notice that unlike our earlier implementations of collections, using a hashing approach results in the access time to a particular element being independent of the number of elements in the table. This means that all of the operations on an element of a hash table should be O(1). This is the result of no longer having to do comparisons to find a particular element or to locate the appropriate position for a given element. Using hashing, we simply calculate where a particular element should be.

KEY CONCEPT
The situation where two elements or keys map to the same location in the table is called a collision.

However, this efficiency is only fully realized if each element maps to a unique position in the table. Consider our example from Figure 14.1. What will happen if we attempt to store the name "Ann" and the name "Andrew"? This situation, where two elements or keys map to the same location in the table, is called a *collision*. We will discuss how to resolve collisions in Section 14.3.

KEY CONCEPT
A hashing function that maps each element to a unique position in the table is said to be a perfect hashing function.

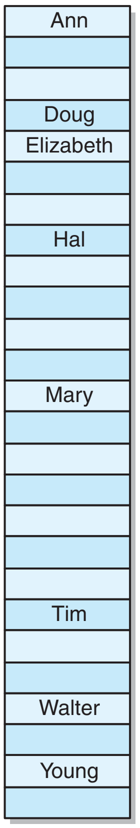


FIGURE 14.1 A simple hashing example

A hashing function that maps each element to a unique position in the table is said to be a *perfect hashing function*. Although it is possible in some situations to develop a perfect hashing function, a hashing function that does a good job of distributing the elements among the table positions will still result in constant time (O(1)) access to elements in the table and an improvement over our earlier algorithms that were either O(n) in the case of our linear approaches or O(log n) in the case of search trees.

Another issue surrounding hashing is the question of how large the table should be. If the data set is of known size and a perfect hashing function can be used, then we simply make the table the same size as the data set. If a perfect hashing function is not available or practical but the size of the data set is known, a good rule of thumb is to make the table 150 percent the size of the data set.

The third case is very common and far more interesting. What if we do not know the size of the data set? In this case, we depend on *dynamic resizing*. Dynamic resizing of a hash table involves creating a new hash table that is larger than, perhaps even twice as large as, the original, inserting all of the elements of the original table into the new table, and then discarding the original table. Deciding when to resize is also an interesting question. One possibility is to use the same method we used with our earlier array implementations and simply expand the table when it is full. However, it is the nature of hash tables that their performance seriously degrades as they become full. A better approach is to use a *load factor*. The load factor of a hash table is the percentage occupancy of the table at which the table will be resized. For example, if the load factor were set to 0.50, then the table would be resized each time it reached 50 percent capacity.

14.2 Hashing Functions

KEY CONCEPT

Extraction involves using only a part of the element's value or key to compute the location at which to store the element.

Although perfect hashing functions are possible if the data set is known, we do not need the hashing function to be perfect to get good performance from the hash table. Our goal is simply to develop a function that does a reasonably good job of distributing our elements in the table such that we avoid collisions. A reasonably good hashing function will still result in constant time access (O(1)) to our data set.

There are a variety of approaches to developing a hashing function for a particular data set. The method that we used in our example in the previous section is called *extraction*. Extraction involves using only a part of the element's value or key to compute the location at which to store the element. In our previous example, we simply extracted the first letter of a string and computed its value relative to the letter A.

Other examples of extraction would be to store phone numbers according to the last four digits or to store information about cars according to the first three characters of the license plate.

The Division Method

Creating a hashing function by *division* simply means we will use the remainder of the key divided by some positive integer p as the index for the given element. This function could be defined as follows:

```
Hashcode(key) = Math.abs(key)%p
```

This function will yield a result in the range from 0 to p–1. If we use our table size as p, we then have an index that maps directly to a location in the table.

Using a prime number p as the table size and the divisor helps provide a better distribution of keys to locations in the table.

For example, if our key value is 79 and our table size is 43, the division method would result in an index value of 36. The division method is very effective when dealing with an unknown set of key values.

The Folding Method

In the *folding method*, the key is divided into parts that are then combined or folded together to create an index into the table. This is done by first dividing the key into parts where each of the parts of the key will be the same length as the desired index, except possibly the last one. In the *shift folding method*, these parts are then added together to create the index. For example, if our key is the Social Security number 987-65-4321, we might divide this into three parts, 987, 654, and 321. Adding these together yields 1962. Assuming we are looking for a three-digit key, at this point we could use either division or extraction to get our index.

KEY CONCEPT

In the shift folding method, the parts of the key are added together to create the index.

A second possibility is *boundary folding*. There are a number of variations on this approach. However, generally, they involve reversing some of the parts of the key before adding. One variation on this approach is to imagine that the parts of the key are written side by side on a piece of paper and that the piece of paper is folded along the boundaries of the parts of the key. In this way, if we begin with the same key, 987-65-4321, we first divide it into parts, 987, 654, and 321. We then reverse every other part of the key, yielding 987, 456, and 321. Adding these together yields 1764 and once again we can proceed with either extraction or division to get our index. Other variations on folding use different algorithms to determine which parts of the key to reverse.

Folding may also be a useful method for building a hashing function for a key that is a string. One approach to this is to divide the string into substrings the same length (in bytes) as the desired index and then combine these strings using an *exclusive-or* function. This is also a useful way to convert a string to a number so that other methods, such as division, may be applied to strings.

The Mid-Square Method

In the *mid-square method*, the key is multiplied by itself, and then the extraction method is used to extract the appropriate number of digits from the middle of the squared result to serve as an index. The same "middle" digits must be chosen each time, to provide consistency. For example, if our key is 4321, we would multiply the key by itself, yielding 18671041. Assuming that we need a three-digit key, we might extract 671 or 710, depending upon how we construct our algorithm. It is also possible to extract bits instead of digits and then construct the index from the extracted bits.

The mid-square method may also be effectively used with strings by manipulating the binary representations of the characters in the string.

The Radix Transformation Method

In the *radix transformation method*, the key is transformed into another numeric base. For example, if our key is 23 in base 10, we might convert it to 32 in base 7. We then use the division method and divide the converted key by the table size and use the remainder as our index. Continuing our previous example, if our table size is 17, we would compute the function:

```
Hashcode(23) = Math.abs(32)%17
             = 15
```

The Digit Analysis Method

In the *digit analysis method*, the index is formed by extracting, and then manipulating, specific digits from the key. For example, if our key is 1234567, we might select the digits in positions 2 through 4, yielding 234, and then manipulate them to form our index. This manipulation can take many forms, including simply reversing the digits (yielding 432), performing a circular shift to the right (yielding 423), performing a circular shift to the left (yielding 342), swapping each pair of digits (yielding 324), or any number of other possibilities, including the methods we have already discussed. The goal is simply to provide a function that does a reasonable job of distributing keys to locations in the table.

The Length-Dependent Method

In the *length-dependent method*, the key and the length of the key are combined in some way to form either the index itself or an intermediate value that is then used with one of our other methods to form the index. For example, if our key is 8765, we might multiply the first two digits by the length and then divide by the last digit, yielding 69. If our table size is 43, we would then use the division method, resulting in an index of 26.

The length-dependent method may also be effectively used with strings by manipulating the binary representations of the characters in the string.

KEY CONCEPT

The length-dependent method and the mid-square method may also be effectively used with strings by manipulating the binary representations of the characters in the string.

Hashing Functions in the Java Language

The `java.lang.Object` class defines a method called `hashcode` that returns an integer based on the memory location of the object. This is generally not very useful. Classes that are derived from `Object` often override the inherited definition of `hashcode` to provide their own version. For example, the `String` and `Integer` classes define their own `hashcode` methods. These more specific `hashcode` functions can be very effective for hashing. By having the `hashcode` method defined in the `Object` class, all Java objects can be hashed. However, it is also possible, and often preferable, to define your own `hashcode` method for any class that you intend to store in a hash table.

KEY CONCEPT
Although Java provides a `hashcode` method for all objects, it is often preferable to define a specific hashing function for any particular class.

14.3 Resolving Collisions

If we are able to develop a perfect hashing function for a particular data set, then we do not need to concern ourselves with collisions, the situation where more than one element or key map to the same location in the table. However, when a perfect hashing function is not possible or practical, there are a number of ways to handle collisions. Similarly, if we are able to develop a perfect hashing function for a particular data set, then we do not need to concern ourselves with the size of the table. In this case, we will simply make the table the exact size of the data set. Otherwise, if the size of the data set is known, it is generally a good idea to set the initial size of the table to about 150 percent of the expected element count. If the size of the data set is not known, then dynamic resizing of the table becomes an issue.

Chaining

The *chaining method* for handling collisions simply treats the hash table conceptually as a table of collections rather than as a table of individual cells. Thus each cell is a pointer to the collection associated with that location in the table. Usually this internal collection is either an unordered list or an ordered list. Figure 14.2 illustrates this conceptual approach.

KEY CONCEPT
The chaining method for handling collisions simply treats the hash table conceptually as a table of collections rather than as a table of individual cells.

Chaining can be implemented in a variety of ways. One approach would be to make the array holding the table larger than the number of cells in the table and use the extra space as an overflow area to store the linked lists associated with each table location. In this method, each position in the array could store both an element (or a key) and the array index of the next

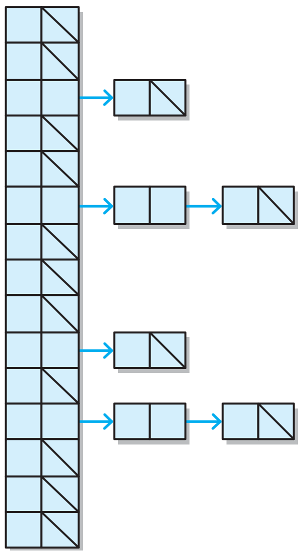


FIGURE 14.2 The chaining method of collision handling

element in its list. The first element mapped to a particular location in the table would actually be stored in that location. The next element mapped to that location would be stored in a free location in this overflow area, and the array index of this second element would be stored with the first element in the table. If a third element is mapped to the same location, the third element would also be stored in this overflow area and the index of third element would be stored with the second element. Figure 14.3 illustrates this strategy.

Note that, using this method, the table itself can never be full. However, if the table is implemented as an array, the array can become full, requiring a decision on whether to throw an exception or simply expand capacity. In our earlier collections, we chose to expand the capacity of the array. In this case, expanding the capacity of the array but leaving the embedded table the original size would have disastrous effects on efficiency. A more complete solution is to expand the array and expand the embedded table within the array. This will, however, require that all of the elements in the table be rehashed using the new table size. We will discuss the dynamic resizing of hash tables further in Section 14.5.

Using this method, the worst case is that our hashing function will not do a good job of distributing elements to locations in the table so that we end up with one linked list of n elements, or a small number of linked lists with roughly n/k elements each, where k is some relatively small constant. In this case, hash tables become O(n) for both insertions and searches. Thus you can see how important it is to develop a good hashing function.

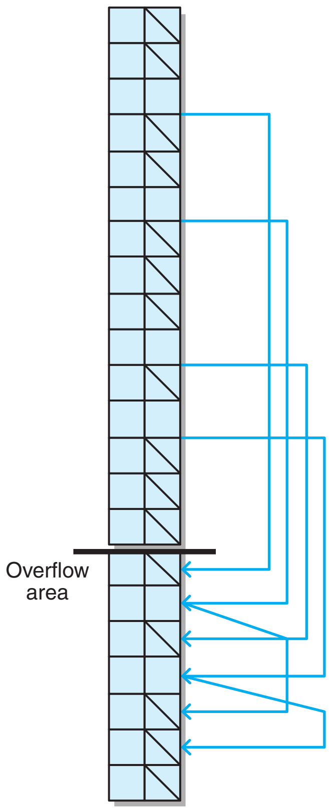


FIGURE 14.3 Chaining using an overflow area

A second method for implementing chaining is to use links. In this method, each cell or bucket in the hash table would be something like the `LinearNode` class used in earlier chapters to construct linked lists. In this way, as a second element is mapped to a particular bucket, we simply create a new `LinearNode`, set the `next` reference of the existing node to point to the new node, set the `element` reference of the new node to the element being inserted, and set the `next` reference of the new node to null. The result is an implementation model that looks exactly like the conceptual model shown in Figure 14.2.

A third method for implementing chaining is to literally make each position in the table a pointer to a collection. In this way, we could represent each position in the table with a list or perhaps even a more efficient collection (e.g., a balanced binary search tree), and this would improve our worst case. Keep in mind, however, that if our hashing function is doing a good job of distributing elements to locations in the table, this approach may incur a great deal of overhead while accomplishing very little improvement.

Open Addressing

The *open addressing method* for handling collisions looks for another open position in the table other than the one to which the element is originally hashed. There are a variety of methods to find another available location in the table. We will examine three of these methods: linear probing, quadratic probing, and double hashing.

KEY CONCEPT

The open addressing method for handling collisions looks for another open position in the table other than the one to which the element is originally hashed.

The simplest of these methods is *linear probing*. In linear probing, if an element hashes to position p and position p is already occupied, we simply try position (p+1)%s, where s is the size of the table. If position (p+1)%s is already occupied, we try position (p+2)%s, and so on until either we find an open position or we find ourselves back at the original position. If we find an open position, we insert the new element. What to do if we do not find an open position is a design decision when creating a hash table. As we have discussed previously, one possibility is to throw an exception if the table is full. A second possibility is to expand the capacity of the table and rehash the existing entries.

The problem with linear probing is that it tends to create clusters of filled positions within the table, and these clusters then affect the performance of insertions and searches. Figure 14.4 illustrates the linear probing method and the creation of a cluster using our earlier hashing function of extracting the first character of the string.

In this example, Ann was entered, followed by Andrew. Because Ann already occupied position 0 of the array, Andrew was placed in position 1. Later, Bob was entered. Because Andrew already occupied position 1, Bob was placed in the next open position, which was position 2. Doug and Elizabeth were already in the table by the time Betty arrived, thus Betty could not be placed in position 1, 2, 3, or 4 and was placed in the next open position, position 5. After Barbara, Hal, and Bill were added, we find that there is now a nine-location cluster at the front of the table, which will continue to grow as more names are added. Thus we see that linear probing may not be the best approach.

A second form of the open addressing method is *quadratic probing*. Using quadratic probing, instead of using a linear approach, once we have a collision, we follow a formula such as

$$\texttt{newhashcode(x)} = \texttt{hashcode(x)} + (-1)^{i-1}((i + 1)/2)^2$$

for i in the range 1 to s–1 where s is the table size.

The result of this formula is the search sequence p, p + 1, p − 1, p + 4, p − 4, p + 9, p − 9, ... Of course, this new hash code is then put through the division method to keep it within the table range. As with linear probing, the same possibility exists that we will eventually get back to the original hash code without having found an open position in which to insert. This "full" condition can be

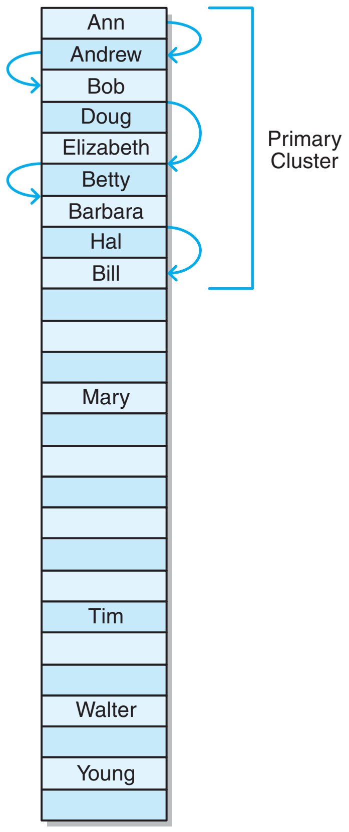


FIGURE 14.4 Open addressing using linear probing

handled in all of the same ways that we described for chaining and linear probing. The benefit of the quadratic probing method is that it does not have as strong a tendency toward clustering as does linear probing. Figure 14.5 illustrates quadratic probing for the same key set and hashing function that we used in Figure 14.4. Notice that after the same data has been entered, we still have a cluster at the front of the table. However, this cluster occupies only six buckets instead of the nine-bucket cluster created by linear probing.

A third form of the open addressing method is *double hashing*. Using the double hashing method, we will resolve collisions by providing a secondary hashing function to be used when the primary hashing function results in a collision. For example, if a key x hashes to a position p that is already occupied, then the next position p′ that we will try will be

```
p' = p + secondaryhashcode(x)
```

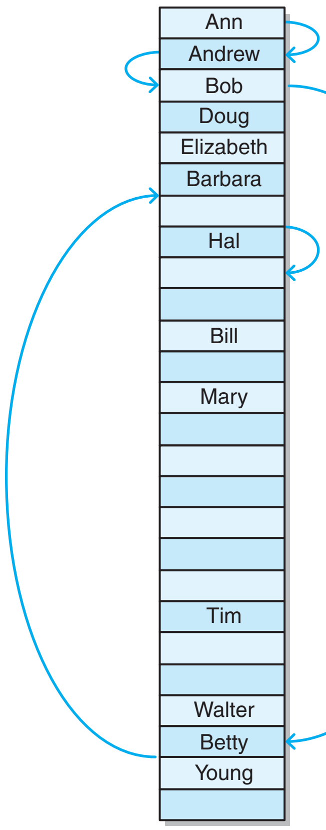


FIGURE 14.5 Open addressing using quadratic probing

If this new position is also occupied, then we look to position

```
p″ = p + 2 * secondaryhashcode(x)
```

We continue searching this way, of course using the division method to maintain our index within the bounds of the table, until an open position is found. This method, while somewhat more costly because of the introduction of an additional function, tends to further reduce clustering beyond the improvement gained by quadratic probing. Figure 14.6 illustrates this approach, again using the same key set and hashing function from our previous examples. For this example, the secondary hashing function is the length of the string. Notice that with the same data, we no longer have a cluster at the front of the table. However, we have

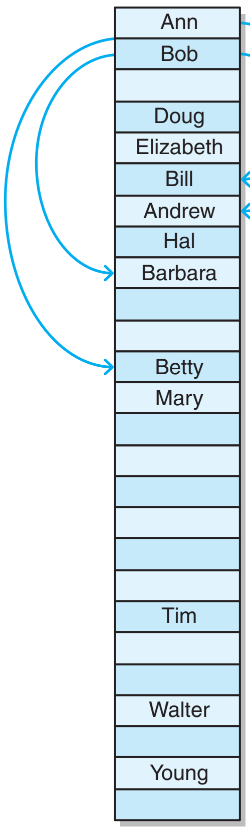


FIGURE 14.6 Open addressing using double hashing

developed a six-bucket cluster from Doug through Barbara. The advantage of double hashing, however, is that even after a cluster has been created, it will tend to grow more slowly than it would if we were using linear probing or even quadratic probing.

14.4 Deleting Elements from a Hash Table

Thus far, our discussion has centered on the efficiency of insertion of and searching for elements in a hash table. What happens if we remove an element from a hash table? The answer to this question depends upon which implementation we have chosen.

Deleting from a Chained Implementation

If we have chosen to implement our hash table using a chained implementation and an array with an overflow area, then removing an element falls into one of five cases:

Case 1 The element we are attempting to remove is the only one mapped to the particular location in the table. In this case, we simply remove the element by setting the table position to null.

Case 2 The element we are attempting to remove is stored in the table (not in the overflow area) but has an index into the overflow area for the next element at the same position. In this case, we replace the element and the next index value in the table with the element and next index value of the array position pointed to by the element to be removed. We then also must set the position in the overflow area to null and add it back to whatever mechanism we are using to maintain a list of free positions.

Case 3 The element we are attempting to remove is at the end of the list of elements stored at that location in the table. In this case, we set its position in the overflow area to null, and we set the next index value of the previous element in the list to null as well. We then also must set the position in the overflow area to null and add it back to whatever mechanism we are using to maintain a list of free positions.

Case 4 The element we are attempting to remove is in the middle of the list of elements stored at that location in the table. In this case, we set its position in the overflow area to null, and we set the next index value of the previous element in the list to the next index value of the element being removed. We then also must add it back to whatever mechanism we are using to maintain a list of free positions.

Case 5 The element we are attempting to remove is not in the list. In this case, we throw an `ElementNotFoundException`.

If we have chosen to implement our hash table using a chained implementation where each element in the table is a collection, then we simply remove the target element from the collection.

Deleting from an Open Addressing Implementation

If we have chosen to implement our hash table using an open addressing implementation, then deletion creates more of a challenge. Consider the example in Figure 14.7. Notice that elements “Ann,” “Andrew,” and “Amy” all mapped to the same location in the table and the collision was resolved using linear probing. What happens if we now remove “Andrew”? If we then search for “Amy” we will not find that element because the search will find “Ann” and then follow the linear probing rule to look in the next position, find it null, and return an exception.

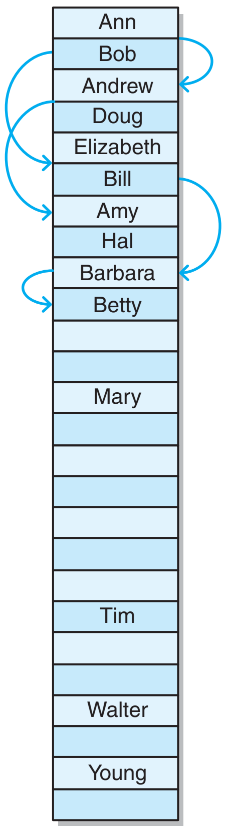


FIGURE 14.7 Open addressing and deletion

The solution to this problem is to mark items as deleted but not actually remove them from the table until some future point when the deleted element is overwritten by a new inserted element or the entire table is rehashed, either because it is being expanded or because we have reached some predetermined threshold for the percentage of deleted records in the table. This means that we will need to add a `boolean` flag to each node in the table and modify all of our algorithms to test and/or manipulate that flag.

14.5 Hash Tables in the Java Collections API

The Java Collections API provides seven implementations of hashing: `Hashtable`, `HashMap`, `HashSet`, `IdentityHashMap`, `LinkedHashSet`, `LinkedHashMap`, and `WeakHashMap`. To understand these different solutions we must first remind ourselves of the distinction between a *set* and a *map* in the Java Collections API as well as some of our other pertinent definitions.

A *set* is a collection of objects where in order to find an object, we must have an exact copy of the object for which we are looking. A *map*, on the other hand, is a collection that stores key-value pairs so that, given the key, we can find the associated value.

KEY CONCEPT
The load factor is the maximum percentage occupancy allowed in the hash table before it is resized.

Another definition that will be useful to us as we explore the Java Collections API implementations of hashing is that of a *load factor*. The load factor, as stated earlier, is the maximum percentage occupancy allowed in the hash table before it is resized. For the implementations that we are going to discuss here, the default is 0.75. Thus, using this default, when one of these implementations becomes 75 percent full, a new hash table is created that is twice the size of the current one, and then all of the elements from the current table are inserted into the new table. The load factor of these implementations can be altered when the table is created.

All of these implementations rely on the `hashcode` method of the object being stored to return an integer. This integer is then processed using the division method (using the table size) to produce an index within the bounds of the table. As stated earlier, the best practice is to define your own `hashcode` method for any class that you intend to store in a hash table.

Let's look at each of these implementations.

The `Hashtable` Class

The `Hashtable` implementation of hashing is the oldest of the implementations in the Java Collections API. In fact, it predates the Collections API and was modified in version 1.2 to implement the `Map` interface so that it would become a part of the Collections API. Unlike the newer Java Collections implementations, `Hashtable` is synchronized. Figure 14.8 shows the operations for the `Hashtable` class.

Creation of a `Hashtable` requires two parameters: initial capacity (with a default of 11) and load factor (with a default of 0.75). Capacity refers to the number of cells or locations in the initial table. Load factor is, as we described earlier, the maximum percentage occupancy allowed in the hash table before it is resized. `Hashtable` uses the chaining method for resolving collisions.

The `Hashtable` class is a legacy class that will be most useful if you are connecting to legacy code or require synchronization. Otherwise, it is preferable to use the `HashMap` class.

Return Value	Method	Description
	`Hashtable()`	Constructs a new, empty hash table with a default initial capacity (11) and load factor, which is 0.75.
	`Hashtable(int initialCapacity)`	Constructs a new, empty hash table with the specified initial capacity and default load factor, which is 0.75.
	`Hashtable(int initialCapacity, float loadFactor)`	Constructs a new, empty hash table with the specified initial capacity and the specified load factor.
	`Hashtable (Map t)`	Constructs a new hash table with the same mappings as the given `Map`.
`void`	`clear()`	Clears this hash table so that it contains no keys.
`Object`	`clone()`	Creates a shallow copy of this hash table.
`boolean`	`contains(Object value)`	Tests if some key maps into the specified value in this hash table.
`boolean`	`containsKey(Object key)`	Tests if the specified object is a key in this hash table.
`boolean`	`containsValue (Object value)`	Returns true if this hash table maps one or more keys to this value.
`Enumeration`	`elements()`	Returns an enumeration of the values in this hash table.
`Set`	`entrySet()`	Returns a `Set` view of the entries contained in this hash table.
`boolean`	`equals(Object o)`	Compares the specified `Object` with this `Map` for equality, as per the definition in the `Map` interface.
`Object`	`get(Object key)`	Returns the value to which the specified key is mapped in this hash table.
`int`	`hashCode()`	Returns the hash code value for this `Map` as per the definition in the `Map` interface.
`boolean`	`isEmpty()`	Tests if this hash table maps no keys to values.
`Enumeration`	`keys()`	Returns an enumeration of the keys in this hash table.
`Set`	`keysSet()`	Returns a `Set` view of the keys contained in this hash table.
`Object`	`put(Object key Object value)`	Maps the specified key to the specified value in this hash table.
`void`	`putAll(Map t)`	Copies all of the mappings from the specified `Map` to this hash table. These mappings will replace any mappings that this hash table had for any of the keys currently in the specified `Map`.
`protected void`	`rehash()`	Increases the capacity of and internally reorganizes this hash table, in order to accommodate and access its entries more efficiently.

FIGURE 14.8 Operations on the `Hashtable` class

`Object`	`remove(Object key)`	Removes the key (and its corresponding value) from this hash table.
`int`	`size()`	Returns the number of keys in this hash table.
`String`	`toString()`	Returns a string representation of this hash table object in the form of a set of entries, enclosed in braces and separated by the ASCII characters comma and space.
`Collection`	`values()`	Returns a `Collection` view of the values contained in this hash table.

FIGURE 14.8 *Continued*

The `HashSet` Class

The `HashSet` class implements the `Set` interface using a hash table. The `HashSet` class, like most of the Java Collections API implementations of hashing, uses chaining to resolve collisions (each table position effectively being a linked list). The `HashSet` implementation does not guarantee the order of the set on iteration and does not guarantee that the order will remain constant over time. This is because the iterator simply steps through the table in order. Because the hashing function will somewhat randomly distribute the elements to table positions, order cannot be guaranteed. Further, if the table is expanded, all of the elements are rehashed relative to the new table size, and the order may change.

Like the `Hashtable` class, the `HashSet` class also requires two parameters: initial capacity and load factor. The default for the load factor is the same as it is for `Hashtable` (0.75). The default for initial capacity is currently unspecified (originally it was 101). Figure 14.9 shows the operations for the `HashSet` class. The `HashSet` class is not synchronized and permits null values.

The `HashMap` Class

The `HashMap` class implements the `Map` interface using a hash table. The `HashMap` class also uses a chaining method to resolve collisions. Like the `HashSet` class, the `HashMap` class is not synchronized and allows null values. Also like the previous implementations, the default load factor is 0.75. Like the `HashSet` class, the current default initial capacity is unspecified though it was also originally 101.

Figure 14.10 shows the operations on the `HashMap` class.

The `IdentityHashMap` Class

The `IdentityHashMap` class implements the `Map` interface using a hash table. The difference between this and the `HashMap` class is that the `IdentityHashMap` class

Return Value	Method	Description
	`HashSet()`	Constructs a new, empty set; the backing `HashMap` instance has the default capacity and load factor, which is 0.75.
	`HashSet(Collection c)`	Constructs a new set containing the elements in the specified collection.
	`HashSet(int initialCapacity)`	Constructs a new, empty set; the backing `HashMap` instance has the specified initial capacity and default load factor, which is 0.75.
	`HashSet(int initial Capacity, float loadFactor)`	Constructs a new, empty set; the backing `HashMap` instance has the specified initial capacity and the specified load factor.
`boolean`	`add(Object o)`	Adds the specified element to this set if it is not already present.
`void`	`clear()`	Removes all of the elements from this set.
`Object`	`clone()`	Returns a shallow copy of this `HashSet` instance: the elements themselves are not cloned.
`boolean`	`contains(Object o)`	Returns true if this set contains the specified element.
`boolean`	`isEmpty()`	Returns true if this set contains no elements.
`iterator()`	`iterator()`	Returns an iterator over the elements in this set.
`boolean`	`remove(Object o)`	Removes the given element from this set if it is present.
`int`	`size()`	Returns the number of elements in this set (its cardinality).

FIGURE 14.9 Operations on the `HashSet` class

uses reference-equality instead of object-equality when comparing both keys and values. This is the difference between using `key1==key2` and using `key1.equals(key2)`.

This class has one parameter: expected maximum size. This is the maximum number of key-value pairs that the table is expected to hold. If the table exceeds this maximum, then the table size will be increased and the table entries rehashed.

Figure 14.11 shows the operations on the `IdentityHashMap` class.

The `WeakHashMap` Class

The `WeakHashMap` class implements the `Map` interface using a hash table. This class is specifically designed with weak keys so that an entry in a `WeakHashMap` will automatically be removed when its key is no longer in use. In other words, if

Return Value	Method	Description
	`HashMap()`	Constructs a new, empty map with a default capacity and load factor, which is 0.75.
	`HashMap(int initialCapacity)`	Constructs a new, empty map with the specified initial capacity and default load factor, which is 0.75.
	`HashMap(int initialCapacity, float loadFactor)`	Constructs a new, empty map with the specified initial capacity and the specified load factor.
	`HashMap(Map t)`	Constructs a new map with the same mappings as the given map.
`void`	`clear()`	Removes all mappings from this map.
`Object`	`clone()`	Returns a shallow copy of this `HashMap` instance: the keys and values themselves are not cloned.
`boolean`	`containsKey(Object key)`	Returns true if this map contains a mapping for the specified key.
`boolean`	`containsValue(Object value)`	Returns true if this map maps one or more keys to the specified value.
`set`	`entrySet()`	Returns a collection view of the mappings contained in this map.
`Object`	`get(Object key)`	Returns the value to which this map maps the specified key.
`boolean`	`isEmpty()`	Returns true if this map contains no key-value mappings.
`Set`	`keySet()`	Returns a set view of the keys contained in this map.
`Object`	`put(Object key, Object value)`	Associates the specified value with the specified key in this map.
`void`	`putAll(Map t)`	Copies all of the mappings from the specified map to this one.
`Object`	`remove(Object key)`	Removes the mapping for this key from this map if present.
`int`	`size()`	Returns the number of key-value mappings in this map.
`Collection`	`values()`	Returns a collection view of the values contained in this map.

FIGURE 14.10 Operations on the `HashMap` class

the use of the key in a mapping in the `WeakHashMap` is the only remaining use of the key, the garbage collector will collect it anyway.

The `WeakHashMap` class allows both null values and null keys, and has the same tuning parameters as the `HashMap` class: initial capacity and load factor.

Figure 14.12 shows the operations on the `WeakHashMap` class.

Return Value	Method	Description
	`IdentityHashMap()`	Constructs a new, empty identity hash map with a default expected maximum size (21).
	`IdentityHashMap(int expectedMaxSize)`	Constructs a new, empty map with the specified expected maximum size.
	`IdentityHashMap(Map m)`	Constructs a new identity hash map containing the key-value mappings in the specified map.
`void`	`clear()`	Removes all mappings from this map.
`Object`	`clone()`	Returns a shallow copy of this identity hash map: the keys and values themselves are not cloned.
`boolean`	`containsKey(Object key)`	Tests whether the specified object reference is a key in this identity hash map.
`boolean`	`containsValue (Object value)`	Tests whether the specified object reference is a value in this identity hash map.
`Set`	`entrySet()`	Returns a set view of the mappings contained in this map.
`boolean`	`equals(Object o)`	Compares the specified object with this map for equality.
`Object`	`get(Object key)`	Returns the value to which the specified key is mapped in this identity hash map, or null if the map contains no mapping for this key.
`int`	`hashCode()`	Returns the hash code value for this map.
`boolean`	`isEmpty()`	Returns true if this identity hash map contains no key-value mappings.
`Set`	`keySet()`	Returns an identity-based set view of the keys contained in this map.
`Object`	`put(Object key, Object value)`	Associates the specified value with the specified key in this identity hash map.
`void`	`putAll(Map t)`	Copies all of the mappings from the specified map to this map. These mappings will replace any mappings that this map had for any of the keys currently in the specified map.
`Object`	`remove(Object key)`	Removes the mapping for this key from this map if present.
`int`	`size()`	Returns the number of key-value mappings in this identity hash map.
`Collection`	`values()`	Returns a collection view of the values contained in this map.

FIGURE 14.11 Operations on the `IdentityHashMap` class

Return Value	Method	Description
	`WeakHashMap()`	Constructs a new, empty `WeakHashMap` with the default initial capacity and the default load factor, which is 0.75.
	`WeakHashMap(int initialCapacity)`	Constructs a new, empty `WeakHashMap` with the given initial capacity and the default load factor, which is 0.75.
	`WeakHashMap(int initial Capacity, float loadFactor)`	Constructs a new, empty `WeakHashMap` with the given initial capacity and the given load factor.
	`WeakHashMap(Map t)`	Constructs a new `WeakHashMap` with the same mappings as the specified map.
`void`	`clear()`	Removes all mappings from this map.
`boolean`	`containsKey(Object key)`	Returns true if this map contains a mapping for the specified key.
`Set`	`entrySet()`	Returns a set view of the mappings in this map.
`Object`	`get(Object key)`	Returns the value to which this map maps the specified key.
`boolean`	`isEmpty()`	Returns true if this map contains no key-value mappings.
`Set`	`keySet()`	Returns a set view of the keys contained in this map.
`Object`	`put(Object key, Object value)`	Associates the specified value with the specified key in this map.
`void`	`putAll(Map t)`	Copies all of the mappings from the specified map to this map. These mappings will replace any mappings that this map had for any of the keys currently in the specified map.
`Object`	`remove(Object key)`	Removes the mapping for the given key from this map, if present.
`int`	`size()`	Returns the number of key-value mappings in this map.
`Collection`	`values()`	Returns a collection view of the values contained in this map.

FIGURE 14.12 Operations on the `WeakHashMap` class

LinkedHashSet and LinkedHashMap

The two remaining hashing implementations are extensions of previous classes. The `LinkedHashSet` class extends the `HashSet` class, and the `LinkedHashMap` class extends the `HashMap` class. Both of them are designed to solve the problem of iterator order. These implementations maintain a doubly linked list running through the entries to maintain the insertion order of the elements. Thus the iterator order for these implementations is the order in which the elements were inserted.

Figure 14.13 shows the additional operations for the `LinkedHashSet` class. Figure 14.14 shows the additional operations for the `LinkedHashMap` class.

Return Value	Method	Description
	`LinkedHashSet()`	Constructs a new, empty linked hash set with the default initial capacity (16) and load factor (0.75).
	`LinkedHashSet (Collection c)`	Constructs a new linked hash set with the same elements as the specified collection.
	`LinkedHashSet (int initialCapacity)`	Constructs a new, empty linked hash set with the specified initial capacity and the default load factor (0.75).
	`LinkedHashSet(int initialCapacity, float loadFactor)`	Constructs a new, empty linked hash set with the specified initial capacity and load factor.

FIGURE 14.13 Additional operations on the `LinkedHashSet` class

Return Value	Method	Description
	`LinkedHashMap()`	Constructs an empty insertion-ordered `LinkedHashMap` instance with a default capacity (16) and load factor (0.75).
	`LinkedHashMap (int initialCapacity)`	Constructs an empty insertion-ordered `LinkedHashMap` instance with the specified initial capacity and a default load factor (0.75).
	`LinkedHashMap (int initialCapacity, float loadFactor)`	Constructs an empty insertion-ordered `LinkedHashMap` instance with the specified initial capacity and load factor.
	`LinkedHashMap (int initialCapacity, float loadFactor, boolean accessOrder)`	Constructs an empty `LinkedHashMap` instance with the specified initial capacity, load factor, and ordering mode.
	`LinkedHashMap(Map m)`	Constructs an insertion-ordered `LinkedHashMap` instance with the same mappings as the specified map.
`void`	`clear()`	Removes all mappings from this map.
`boolean`	`containsValue (Object value)`	Returns true if this map maps one or more keys to the specified value.
`Object`	`get(Object key)`	Returns the value to which this map maps the specified key.
`protected boolean`	`removeEldestEntry (Map.Entry eldest)`	Returns true if this map should remove its eldest entry.

FIGURE 14.14 Additional operations on the `LinkedHashMap` class

Summary of Key Concepts

- In hashing, elements are stored in a hash table, with their location in the table determined by a hashing function.
- The situation where two elements or keys map to the same location in the table is called a collision.
- A hashing function that maps each element to a unique position in the table is said to be a perfect hashing function.
- Extraction involves using only a part of the element's value or key to compute the location at which to store the element.
- The division method is very effective when dealing with an unknown set of key values.
- In the shift folding method, the parts of the key are added together to create the index.
- The length-dependent method and the mid-square method may also be effectively used with strings by manipulating the binary representations of the characters in the string.
- Although Java provides a `hashcode` method for all objects, it is often preferable to define a specific hashing function for any particular class.
- The chaining method for handling collisions simply treats the hash table conceptually as a table of collections rather than as a table of individual cells.
- The open addressing method for handling collisions looks for another open position in the table other than the one to which the element is originally hashed.
- The load factor is the maximum percentage occupancy allowed in the hash table before it is resized.

Self-Review Questions

SR 14.1 What is the difference between a hash table and the other collections we have discussed?

SR 14.2 What is a collision in a hash table?

SR 14.3 What is a perfect hashing function?

SR 14.4 What is our goal for a hashing function?

SR 14.5 What is the consequence of not having a good hashing function?

SR 14.6 What is the extraction method?

SR 14.7 What is the division method?

SR 14.8 What is the shift folding method?

SR 14.9 What is the boundary folding method?

SR 14.10 What is the mid-square method?

SR 14.11 What is the radix transformation method?

SR 14.12 What is the digit analysis method?

SR 14.13 What is the length-dependent method?

SR 14.14 What is chaining?

SR 14.15 What is open addressing?

SR 14.16 What are linear probing, quadratic probing, and double hashing?

SR 14.17 Why is deletion from an open addressing implementation a problem?

SR 14.18 What is the load factor, and how does it affect table size?

Exercises

EX 14.1 Draw the hash table that results from adding the following integers (34 45 3 87 65 32 1 12 17) to a hash table of size 11 using the division method and linked chaining.

EX 14.2 Draw the hash table from Exercise 14.1 using a hash table of size 11 using array chaining with a total array size of 20.

EX 14.3 Draw the hash table from Exercise 14.1 using a table size of 17 and open addressing using linear probing.

EX 14.4 Draw the hash table from Exercise 14.1 using a table size of 17 and open addressing using quadratic probing.

EX 14.5 Draw the hash table from Exercise 14.1 using a table size of 17 and double hashing using extraction of the first digit as the secondary hashing function.

EX 14.6 Draw the hash table that results from adding the following integers (1983, 2312, 6543, 2134, 3498, 7654, 1234, 5678, 6789) to a hash table using shift folding of the first two digits with the last two digits. Use a table size of 13.

EX 14.7 Draw the hash table from Exercise 14.6 using boundary folding.

EX 14.8 Draw a UML diagram that shows how all of the various implementations of hashing within the Java Collections API are constructed.

Programming Projects

PP 14.1 Implement the hash table illustrated in Figure 14.1 using the array version of chaining.

PP 14.2 Implement the hash table illustrated in Figure 14.1 using the linked version of chaining.

PP 14.3 Implement the hash table illustrated in Figure 14.1 using open addressing with linear probing.

PP 14.4 Implement a dynamically resizable hash table to store people's names and Social Security numbers. Use the extraction method with division using the last four digits of the Social Security number. Use an initial table size of 31 and a load factor of 0.80. Use open addressing with double hashing using an extraction method on the first three digits of the Social Security number.

PP 14.5 Implement the problem from Programming Project 14.4 using linked chaining.

PP 14.6 Implement the problem from Programming Project 14.4 using the `HashMap` class of the Java Collections API.

PP 14.7 Create a new implementation of the bag collection called `HashtableBag` using a hash table.

PP 14.8 Implement the problem from Programming Project 14.4 using shift folding with the Social Security number divided into three equal three-digit parts.

PP 14.9 Create a graphical system that will allow a user to add and remove employees where each employee has an employee id (six-digit number), employee name, and years of service. Use the `hashcode` method of the `Integer` class as your hashing function and use one of the Java Collections API implementations of hashing.

PP 14.10 Complete Programming Project 14.9 using your own `hashcode` function. Use extraction of the first three digits of the employee id as the hashing function and use one of the Java Collections API implementations of hashing.

PP 14.11 Complete Programming Project 14.9 using your own `hashcode` function and your own implementation of a hash table.

PP 14.12 Create a system that will allow a user to add and remove vehicles from an inventory system. Vehicles will be represented by license number (an eight-character string), make, model, and color. Use your own array-based implementation of a hash table using chaining.

PP 14.13 Complete Programming Project 14.12 using a linked implementation with open addressing and double hashing.

Answers to Self-Review Questions

SRA 14.1 Elements are placed into a hash table at an index produced by a function of the value of the element or a key of the element. This is unique from other collections where the position/location of an element in the collection is determined either by comparison with the other values in the collection or by the order in which the elements were added or removed from the collection.

SRA 14.2 The situation where two elements or keys map to the same location in the table is called a collision.

SRA 14.3 A hashing function that maps each element to a unique position in the table is said to be a perfect hashing function.

SRA 14.4 We need a hashing function that will do a good job of distributing elements into positions in the table.

SRA 14.5 If we do not have a good hashing function, the result will be too many elements mapped to the same location in the table. This will result in poor performance.

SRA 14.6 Extraction involves using only a part of the element's value or key to compute the location at which to store the element.

SRA 14.7 The division method involves dividing the key by some positive integer p (usually the table size and usually prime) and then using the remainder as the index.

SRA 14.8 Shift folding involves dividing the key into parts (usually the same length as the desired index) and then adding the parts. Extraction or division is then used to get an index within the bounds of the table.

SRA 14.9 Like shift folding, boundary folding involves dividing the key into parts (usually the same length as the desired index). However, some of the parts are then reversed before adding. One example is to imagine that the parts are written side by side on a piece of paper, which is then folded on the boundaries between parts. In this way, every other part is reversed.

SRA 14.10 The mid-square method involves multiplying the key by itself and then extracting some number of digits or bytes from the middle of the result. Division can then be used to guarantee an index within the bounds of the table.

SRA 14.11 The radix transformation method is a variation on the division method where the key is first converted to another numeric base and then divided by the table size with the remainder used as the index.

SRA 14.12 In the digit analysis method, the index is formed by extracting, and then manipulating, specific digits from the key.

SRA 14.13 In the length-dependent method, the key and the length of the key are combined in some way to form either the index itself or an intermediate value that is then used with one of our other methods to form the index.

SRA 14.14 The chaining method for handling collisions simply treats the hash table conceptually as a table of collections rather than as a table of individual cells. Thus each cell is a pointer to the collection associated with that location in the table. Usually this internal collection is either an unordered list or an ordered list.

SRA 14.15 The open addressing method for handling collisions looks for another open position in the table other than the one to which the element is originally hashed.

SRA 14.16 Linear probing, quadratic probing, and double hashing are methods for determining the next table position to try if the original hash causes a collision.

SRA 14.17 Because of the way that a path is formed in open addressing, deleting an element from the middle of that path can cause elements beyond that on the path to be unreachable.

SRA 14.18 The load factor is the maximum percentage occupancy allowed in the hash table before it is resized. Once the load factor has been reached, a new table is created that is twice the size of the current table, and then all of the elements in the current table are inserted into the new table.

15 Sets and Maps

This chapter wraps up our discussion of collections by introducing both set and map collections. These are the collections used in the Java Collections API to represent many of the collections presented in this text. We will introduce our own implementations and also discuss the Java Collections API implementations.

CHAPTER OBJECTIVES

- Define set and map collections
- Use a set collection to solve a problem
- Examine an array implementation of a set
- Examine a linked implementation of a set
- Explore Java Collections API implementations of sets and maps

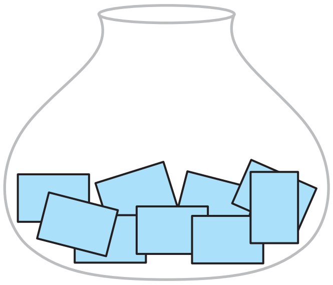


FIGURE 15.1 The conceptual view of a set collection

15.1 A Set Collection

A *set* can be defined as a collection of elements with no duplicates. For our current purposes, we will assume that there is no particular positional relationship among the elements of the set. Conceptually it is similar to a bag or box into which elements are placed. Figure 15.1 depicts a set collection holding its otherwise unorganized elements.

KEY CONCEPT
A set is a nonlinear collection in which there is essentially no inherent organization to the elements in the collection.

A set is a nonlinear collection. There is essentially no organization to the elements in the collection at all. The elements in a set have no inherent relationship to each other, and there is no significance to the order in which they have been added to the set.

The operations we define for a set collection are listed in Figure 15.2. Like our other collections, a set has operations that allow the user to

Operation	Description
`add`	Adds an element to the set.
`addAll`	Adds the elements of one set to another.
`removeRandom`	Removes an element at random from the set.
`remove`	Removes a particular element from the set.
`union`	Combines the elements of two sets to create a third.
`contains`	Determines if a particular element is in the set.
`equals`	Determines if two sets contain the same elements.
`isEmpty`	Determines if the set is empty.
`size`	Determines the number of elements in the set.
`iterator`	Provides an iterator for the set.
`toString`	Provides a string representation of the set.

FIGURE 15.2 The operations on a set collection

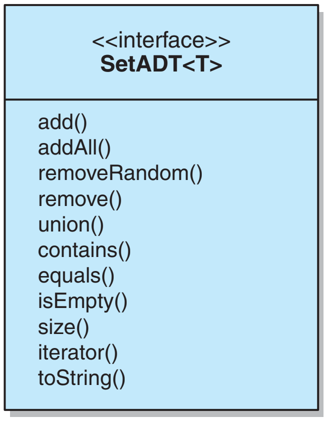


FIGURE 15.3 UML description of the `SetADT<T>` interface

add and remove elements. Some operations such as `isEmpty` and `size` are common to almost all collections as well. A set collection is somewhat unique in that it incorporates an element of randomness. The very nature of a set collection lends itself to being able to pick an element out of the set at random.

Listing 15.1 defines a Java interface for a set collection. Figure 15.3 illustrates the UML description of the `SetADT` interface.

LISTING 15.1

```
/**
 * SetADT defines the interface to a set collection.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 9/21/2008
 */

package jss2;
import java.util.Iterator;

public interface SetADT<T>
{
   /**
    * Adds one element to this set, ignoring duplicates.
    *
    * @param element  the element to be added to this set
    */
  public void add (T element);
```

LISTING 15.1 *continued*

```
/**
 * Removes and returns a random element from this set.
 *
 * @return  a random element from this set
 */
public T removeRandom ();

/**
 * Removes and returns the specified element from this set.
 *
 * @param element  the element to be removed from this list
 * @return         the element just removed from this list
 */
public T remove (T element);

/**
 * Returns the union of this set and the parameter
 *
 * @param set  the set to be unioned with this set
 * @return     a set that is the union of this set and the parameter
 */
public SetADT<T> union (SetADT<T> set);

/**
 * Returns true if this set contains the parameter
 *
 * @param target   the element being sought in this set
 * @return         true if this set contains the parameter
 */
public boolean contains (T target);

/**
 * Returns true if this set and the parameter contain exactly
 * the same elements
 *
 * @param set  the set to be compared with this set
 * @return     true if this set and the parameter contain exactly
 *             the same elements
 */
public boolean equals (SetADT<T> set);

/**
 * Returns true if this set contains no elements
 *
```

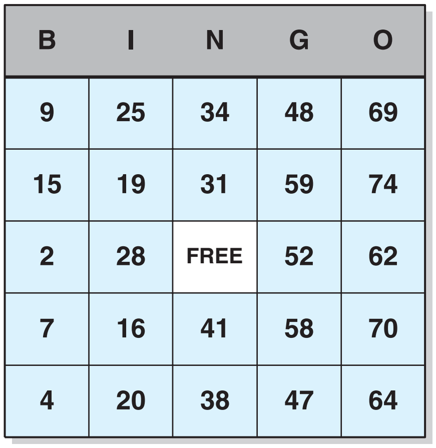
LISTING 15.1 *continued*

```
     * @return  true if this set contains no elements
     */
    public boolean isEmpty();

    /**
     * Returns the number of elements in this set
     *
     * @return  the integer number of elements in this set
     */
    public int size();

    /**
     * Returns an iterator for the elements in this set
     *
     * @return  an iterator for the elements in this set
     */
    public Iterator<T> iterator();

    /**
     * Returns a string representation of this set
     *
     * @return  a string representation of this set
     */
    public String toString();
}
```

Note that because there is no inherent order in a set, the order of the elements returned by the iterator method will be arbitrary.

15.2 Using a Set: Bingo

We can use the game called bingo to demonstrate the use of a set collection. In bingo, numbers are chosen at random from a limited set, usually 1 to 75. The numbers in the range 1 to 15 are associated with the letter B, 16 to 30 with the letter I, 31 to 45 with the letter N, 46 to 60 with the letter G, and 61 to 75 with the letter O. The person managing the game (the "caller") selects a number randomly, and then announces the letter and the number. The caller then sets aside that number so that it cannot be used again in that game. All of the players then mark any squares on their card that match the letter and number called. Once any



B	I	N	G	O
9	25	34	48	69
15	19	31	59	74
2	28	FREE	52	62
7	16	41	58	70
4	20	38	47	64

FIGURE 15.4 A bingo card

player has five squares in a row marked (vertically, horizontally, or diagonally), they announce "Bingo!" and claim their prize. Figure 15.4 shows a sample bingo card.

A set is perfectly suited for assisting the caller in selecting randomly from the possible numbers. To solve this problem, we would simply need to create an object for each of the possible numbers and add them to a set. Then, each time the caller needs to select a number, we would call the `removeRandom` method. Listing 15.2 shows the `BingoBall` class needed to represent each possible selection. The program in Listing 15.3 adds the 75 bingo balls to the set and then selects some of them randomly to illustrate the task.

In the `Bingo` program, the set is represented as an object of type `ArraySet`. More specifically, it is an object of type `ArraySet<BingoBall>`, a set that stores `BingoBall` objects. We explore the implementation of the `ArraySet` class in the next section.

Figure 15.5 shows the relationship between the `Bingo` and `BingoBall` classes illustrated in UML.

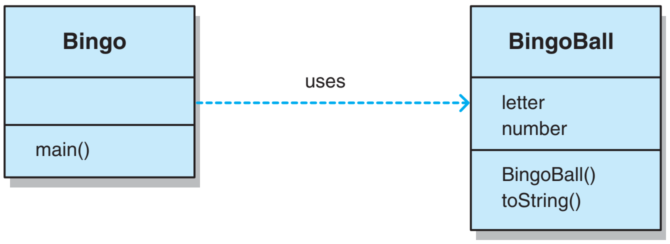


FIGURE 15.5 UML description of the `Bingo` and `BingoBall` classes

Listing 15.2

```java
/**
 * BingoBall represents a ball used in a Bingo game.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 9/21/2008
 */

public class BingoBall
{
  private char letter;
  private int number;

  /**
   * Sets up this Bingo ball with the specified number and the
   * appropriate letter.
   *
   * @param num  the number to be applied to the new bingo ball
   */
  public BingoBall (int num)
  {
    number = num;

    if (num <= 15)
      letter = 'B';
    else
      if (num <= 30)
        letter = 'I';
      else
        if (num <= 45)
          letter = 'N';
        else
          if (num <= 60)
            letter = 'G';
          else
            letter = 'O';
  }

  /**
   * Returns a string representation of this bingo ball.
   *
   * @return  a string representation of the bingo ball
  */
```

LISTING 15.2 *continued*

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

LISTING 15.3

```
/**
 * Bingo demonstrates the use of a set collection.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 9/21/2008
 */
import jss2.ArraySet;

public class Bingo
{

  /**
   * Creates all 75 Bingo balls and stores them in a set. Then
   * pulls several balls from the set at random and prints them.
   */
  public static void main (String[] args)
  {
    final int NUM_BALLS = 75, NUM_PULLS = 10;
    ArraySet<BingoBall> bingoSet = new ArraySet<BingoBall>();
    BingoBall ball;

    for (int num = 1; num <= NUM_BALLS; num++)
    {
      ball = new BingoBall (num);
      bingoSet.add (ball);
    }

    System.out.println ("Size: " + bingoSet.size());
    System.out.println ();
```

LISTING 15.3 *continued*

```
    for (int num = 1; num <= NUM_PULLS; num++)
    {
      ball = bingoSet.removeRandom();
      System.out.println (ball);
    }
  }
}
```

15.3 Implementing a Set: With Arrays

So far in our discussion of a set collection we have described its basic conceptual nature and the operations that allow the user to interact with it. In software engineering terms, we would say that we have done the analysis for a set collection. We have also used a set, without knowing the details of how it was implemented, to solve a particular problem. Now let's turn our attention to the implementation details. There are various ways to implement a class that represents a set. In this section, we examine an implementation strategy that uses an array to store the objects contained in the set. In the next section, we examine a linked implementation of a set.

KEY CONCEPT
The implementation of the collection operations should not affect the way users interact with the collection.

The key instance data for the `ArraySet<T>` class includes the array that holds the contents of the set and the integer variable `count` that keeps track of the number of elements in the collection. We also define a `Random` object to support the drawing of a random element from the set, and a constant, `DEFAULT_CAPACITY`, to define a default capacity. We create another constant, `NOT_FOUND`, to assist in the operations that search for particular elements. The data associated with the `ArraySet<T>` class is declared as follows:

```
/**
 * ArraySet represents an array implementation of a set.
 *
 * @author Dr. Lewis
 * @author Dr. Chase
 * @version 1.0, 9/21/2008
 */
```

```
package jss2;
import jss2.exceptions.*;
import java.util.*;

public class ArraySet<T> implements SetADT<T>, Iterable<T>
{
  private static Random rand = new Random();
  private final int DEFAULT_CAPACITY = 100;
  private final int NOT_FOUND = -1;
  private int count;  // the current number of elements in the set
  private T[] contents;
```

Note that the `Random` object is declared as a static variable and is instantiated in its declaration (rather than in a constructor). Because it is static, the `Random` object is shared among all instances of the `ArraySet<T>` class. This strategy avoids the problem of creating two sets that have random-number generators using the same seed value.

Like we did with the stack implementation, the value of the variable `count` actually represents two related pieces of information. First, it represents the number of elements that are currently stored in the set collection. Second, because Java array indexes start at zero, it also represents the next open slot into which a new element can be stored in the array. On one hand the value of `count` represents the abstract state of the collection, and on the other it helps us with the internal implementation of that collection.

In this implementation, the elements contained in the set are gathered contiguously at one end of the array. This strategy simplifies various aspects of the operations, though it does require operations that remove elements to "fill in the gaps" created in the elements. Figure 15.6 depicts the use of an array to store the elements of a set.

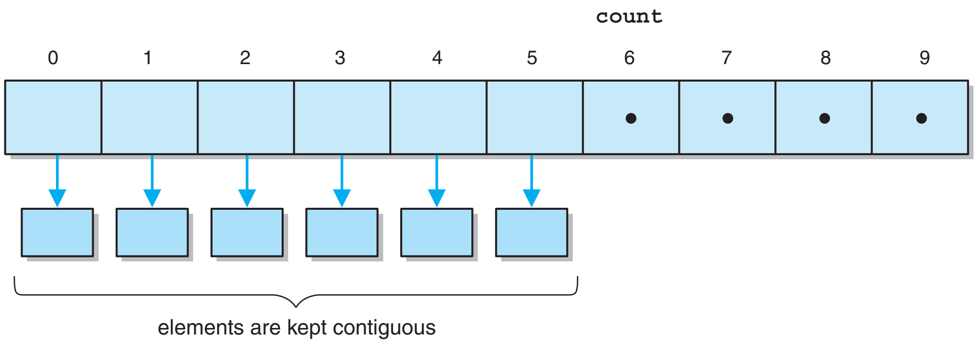


FIGURE 15.6 An array implementation of a set collection

The following constructor is defined for the ArraySet<T> class to set up an initially empty set. The value of count is set to zero, and the array that will store the elements of the set is instantiated. This constructor uses a default value for the initial capacity of the contents array.

```
/**
 * Creates an empty set using the default capacity.
 */
public ArraySet()
{
  count = 0;
  contents = (T[])(new Object[DEFAULT_CAPACITY]);
}
```

We also provide a second constructor that accepts a single integer parameter representing the initial capacity of the contents array. In a particular situation, the user may know approximately how many elements will be stored in a set and can specify that value from the beginning. This overloaded constructor can be defined as follows:

```
/**
 * Creates an empty set using the specified capacity.
 *
 * @param initialCapacity   the initial capacity for the array set
 */
public ArraySet (int initialCapacity)
{
  count = 0;
  contents = (T[])(new Object[initialCapacity]);
}
```

As we design the implementation of each operation, we must consider any situations that may exist that would require any special processing. For example, if the collection is empty, an element cannot be removed from it. Likewise, because we are dealing with a fixed capacity, we must consider the situation in which the underlying data structure is full.

KEY CONCEPT
In the Java Collections API and throughout this text, class names indicate both the underlying data structure and the collection.

The add Operation

The purpose of the add method is to incorporate into the set collection the T object that it accepts as a parameter. For our array implementation, that means storing the T object in an empty slot in the array. The instance variable count represents the next empty space in the array, so we can simply store the new element at that location.

However, the add operation for a fixed-capacity structure must take into account the situation in which the array is filled. As with our previous array implementations, our solution is to automatically expand the capacity of the array when this situation arises.

The following method implements the add operation for the ArraySet<T> class:

```
/**
 * Adds the specified element to the set if it is not already
 * present. Expands the capacity of the set array if necessary.
 *
 * @param element  the element to be added to the set array
 */
public void add (T element)
{
  if (!(contains(element)))
  {
    if (size() == contents.length)
      expandCapacity();

    contents[count] = element;
    count++;
  }
}
```

First, the add method uses the contains method to determine if a duplicate value already exists in the array. If this is the case, the method has no effect on the collection.

The add method then uses the size method to determine the number of elements currently in the collection. If this value equals the total number of cells in the array, indicated by the length constant, then the expandCapacity method is called. Regardless of whether or not the capacity is expanded, the element is then stored in the array and the number of elements in the set collection is incremented. Note that after the add method finishes, the value of the count variable continues to represent both the number of elements in the set and the next open slot in the array.

Instead of calling the size method, the add method could have examined the value of the count variable to determine if the capacity of the array needed to be expanded. The value of count is, after all, exactly what the size method returns. However, in situations like this in which there is a method to play a particular role (determine the size), we are better off using this method. If the design is later changed to determine the size of the set in a different way, the `add` method would still work without a problem.

The `expandCapacity` method increases the size of the array that is storing the elements of the set. More precisely, it creates a second array that is twice the size of the one currently storing the contents of the set, copies all of the current references into the new array, then resets the `contents` instance variable to refer to the larger array. The `expandCapacity` method is implemented as follows:

```
/**
 * Creates a new array to store the contents of the set with
 * twice the capacity of the old one.
 */
private void expandCapacity()
{
  T[] larger = (T[])(new Object[contents.length*2]);

  for (int index=0; index < contents.length; index++)
    larger[index] = contents[index];

  contents = larger;
}
```

Note that the `expandCapacity` method is declared with private visibility. It is designed as a support method, not as a service provided for the user of the set collection.

Also note that the `expandCapacity` method doubles the size of the `contents` array. It could have tripled the size, or simply added ten more cells, or even just one. The amount of the increase determines how soon we will have to increase the size again. We don't want to have to call the `expandCapacity` method too often, because it copies the entire contents of the collection from one array to another. We also don't want to have too much unused space in the array, even though this is probably the less serious offense. There is some mathematical analysis that could be done to determine the most effective size increase, but at this point we will simply make reasonable choices.

Like our other array implementations, the cost of expanding the capacity of the array is amortized over all of the calls to the `add` method. Thus, because the add method consists only of simple assignment statements, it is O(1).

The `addAll` Operation

The purpose of the `addAll` method is to incorporate all of the objects from one set, which it accepts as a parameter, into the set collection. For our array implementation, this means that we can use our `iterator` method to step through the contents of one set and use our `add` method to add those elements to the current set. One advantage of using the `add` method in this way is that the `add` method already checks capacity and expands the array if necessary. Because we have already determined that our `add` method is O(1), we need only concern ourselves with the loop using the iterator. This loop will execute n times, once for each element in the given set. Thus our `addAll` operation is O(n).

The following method implements the `addAll` operation for the `ArraySet<T>` class:

```
/**
 * Adds the contents of the parameter to this set.
 *
 * @param set  the collection to be added to this set
 */
public void addAll (SetADT<T> set)
{
  Iterator<T> scan = set.iterator();

  while (scan.hasNext())
    add (scan.next());
}
```

The `removeRandom` Operation

The `removeRandom` operation must choose an element from the collection at random, remove that element from the collection, and return it to the calling method. This operation relies on the static `Random` object called `rand` that is defined at the class level.

The only special case for this operation is when an attempt is made to remove an element from an empty set. If the set collection is empty, this method throws an `EmptyCollectionException`. This processing is consistent with our philosophy of using exceptions.

The `removeRandom` method of the `ArraySet<T>` class is written as follows:

```
/**
 * Removes a random element from the set and returns it. Throws
 * an EmptyCollectionException if the set is empty.
 *
```

```
     * @return                    a random element from the set
     * @throws EmptyCollectionException  if an empty set exception occurs
     */
    public T removeRandom() throws EmptyCollectionException
    {
      if (isEmpty())
        throw new EmptyCollectionException("Stack");

      int choice = rand.nextInt(count);

      T result = contents[choice];

      contents[choice] = contents[count-1];  // fill the gap
      contents[count-1] = null;
      count--;

      return result;
    }
```

The `nextInt` method of the `Random` class is used to determine a pseudorandom value in the range from 0 to `count-1`. This range represents the indices of all elements currently stored in the array. Once the random element is chosen, it is stored in the local variable called `result`, which is returned to the calling method when this method is complete.

Recall that this implementation of the set collection keeps all elements in the set stored contiguously at one end of the `contents` array. Because this method removes one of the elements, we must "fill the gap" in some way. We could use a loop to shift all of the elements down one, but that is unnecessary. Because there is no ordering implied by the array, we can simply take the last element in the list (at index `count-1`) and put it in the cell of the removed element, which requires no looping. Thus our `removeRandom` method is O(1). Were we to alter the algorithm to shift elements down instead, our method would be O(n).

The remove Operation

The `remove` operation removes the specified element from the set and returns it. This method will throw an `EmptyCollectionException` if the set is empty and a `NoSuchElementException` if the target element is not in the set.

```
/**
 * Removes the specified element from the set and returns it.
 * Throws an EmptyCollectionException if the set is empty and a
 * NoSuchElementException if the target is not in the set.
 *
 * @param target                   the element being sought in the set
 * @return                         the element specified by the target
 * @throws EmptyCollectionException       if an empty set exception occurs
 * @throws NoSuchElementException  if a no such element exception occurs
 */
public T remove (T target) throws EmptyCollectionException,
                                   NoSuchElementException
{
  int search = NOT_FOUND;

  if (isEmpty())
    throw new EmptyCollectionException("Stack");

  for (int index=0; index < count && search == NOT_FOUND; index++)
    if (contents[index].equals(target))
      search = index;

  if (search == NOT_FOUND)
    throw new NoSuchElementException();

  T result = contents[search];

  contents[search] = contents[count-1];
  contents[count-1] = null;
  count--;

  return result;
}
```

With our array implementation, the `remove` operation is simply a matter of searching the array for the target element, removing it, and replacing it with the element stored at `count-1`, or the last element stored in the array. Because the elements of a set are not stored in any particular order, there is no need to shift more than the one element. We then decrement the count. Because of the need to do a linear search of the array, this method is O(n).

The `union` Operation

The `union` operation returns a new set that is the union of this set and the parameter; i.e., a new set that contains all of the elements from both sets. Again, we can use our existing operations. We use our constructor to create a new set and then step through our array and use the `add` method to add each element of our current set to the new set. Next, we create an iterator for the set passed as a parameter, step through each element of that set, and add each of them to the new set. Because there is no inherent order in a set, it does not matter which set's contents we add first.

There are several interesting design possibilities with this operation. First, because the method is returning a new set that is the combination of this set and another set, one could argue that the method should simply be a static method accepting two sets as input parameters. For consistency, we have chosen not to use that solution. A second possibility is to use the `addAll` method (i.e., `both.addAll(this)` followed by `both.addAll(set)`). However, we have deliberately chosen to use a `for` loop and an iterator in this implementation to demonstrate an important concept. Because the process occurs "inside" one set, we have access to its private instance data and thus can use a `for` loop to traverse the array. However, for the set passed as a parameter, we use an iterator to access its elements. Because of the two linear loops, this method is O(n + m) where n and m are the sizes of the two sets.

```
/**
 * Returns a new set that is the union of this set and the
 * parameter.
 *
 * @param set  the set that is to be unioned with this set
 * @return     a new set that is the union of this set and
 *             the parameter
 */
public SetADT<T> union (SetADT<T> set)
{
  ArraySet<T> both = new ArraySet<T>();

  for (int index = 0; index < count; index++)
    both.add (contents[index]);

  Iterator<T> scan = set.iterator();
  while (scan.hasNext())
    both.add (scan.next());

  return both;
}
```

The `contains` Operation

The `contains` operation returns true if this set contains the specified target element. As with the `remove` operation, because of our array implementation, this operation becomes a simple search of an array to locate a particular element and is O(n).

```
/**
 * Returns true if this set contains the specified target
 * element.
 *
 * @param target  the element being sought within this set
 * @return        true if the set contains the target element
 */
public boolean contains (T target)
{
  int search = NOT_FOUND;

  for (int index=0; index < count && search == NOT_FOUND; index++)
    if (contents[index].equals(target))
      search = index;

  return (search != NOT_FOUND);
}
```

The `equals` Operation

The `equals` operation will return true if the current set contains exactly the same elements as the set passed as a parameter. If the two sets are of different sizes, then there is no reason to continue the comparison. However, if the two sets are the same size, we create a deep copy of each set and then use an iterator to step through the elements of the set passed as a parameter and use the `contains` method to confirm that each of those elements is also in the current set. As we find elements in both sets, we remove them from the copies, being careful not to affect the original sets. If both of the copies are empty at the end of the process, then the sets are indeed equal. Notice that we iterate over the original set passed as a parameter while removing matching elements from the copies. This avoids any problems associated with modifying a set while using the associated iterator. Because we are iterating over one of the sets, this operation is O(n).

```
/**
 * Returns true if this set contains exactly the same elements
 * as the parameter.
 *
 * @param set  the set to be compared with this set
```

```
     * @return    true if the parameter set and this set contain
     *             exactly the same elements
     */
    public boolean equals (SetADT<T> set)
    {
      boolean result = false;
      ArraySet<T> temp1 = new ArraySet<T>();
      ArraySet<T> temp2 = new ArraySet<T>();
      T obj;

      if (size() == set.size())
      {
        temp1.addAll(this);
        temp2.addAll(set);

        Iterator<T> scan = set.iterator();

        while (scan.hasNext())
        {
          obj = scan.next();
          if (temp1.contains(obj))
          {
            temp1.remove(obj);
            temp2.remove(obj);
          }
        }
        result = (temp1.isEmpty() && temp2.isEmpty());
      }
      return result;
    }
```

Other Operations

The `iterator`, `toString`, `size` and `isEmpty` methods can be implemented using the same strategies used in earlier implmentations. These methods are left as programming projects.

UML Description

Now that we have all of our classes defined, it is possible to see a UML representation of the entire class diagram for our Bingo example, as illustrated in Figure 15.7.

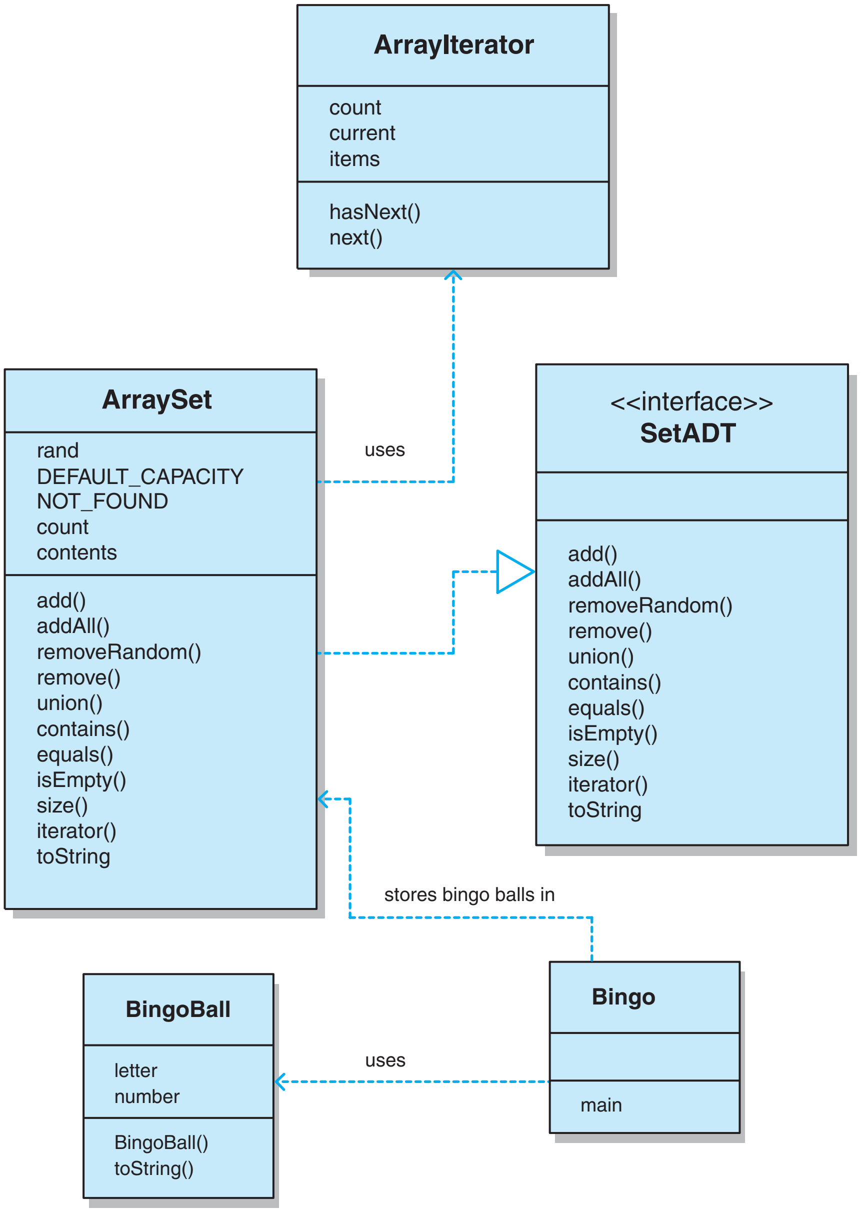


FIGURE 15.7 UML description of the bingo system

15.4 Implementing a Set: With Links

The LinkedSet<T> class implements the SetADT<T> interface. Because we are using a linked list approach, we need only a single reference to the first node in the list. We will also maintain a count of the number of elements in the list. Finally, we need a Random object to support the removeRandom operation. We can also reuse our LinearNode class that we used with stacks, queues, and lists. The class-level data of the LinkedSet<T> class is therefore:

```
/**
 * LinkedSet represents a linked implementation of a set.
 *
 * @author Dr. Chase
 * @author Dr. Lewis
 * @version 1.0, 9/21/2008
 */
package jss2;
import jss2.exceptions.*;
import java.util.*;

public class LinkedSet<T> implements SetADT<T>, Iterable<T>
{
  private static Random rand = new Random();
  private int count;  // the current number of elements in the set
  private LinearNode<T> contents;
```

Using the LinearNode<T> class and maintaining a count of elements in the collection creates the implementation strategy depicted in Figure 15.8.

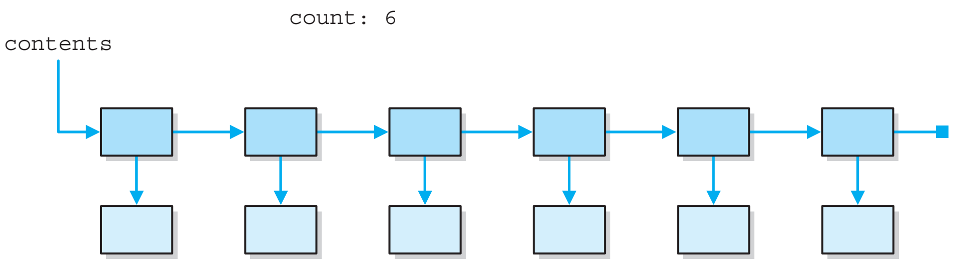


FIGURE 15.8 A linked implementation of a set collection

The constructor of the `LinkedSet<T>` class sets the count of elements to zero and sets the front of the list, represented by the variable `contents`, to `null`.

```
/**
 * Creates an empty set.
 */
public LinkedSet()
{
  count = 0;
  contents = null;
}
```

The add Operation

The `add` method incorporates the element passed as a parameter into the collection. Because there is no inherent order among the elements of a set, we can simply add the new element to the front of the list after verifying that there are no duplicates:

```
/**
 * Adds the specified element to this set if it is not already
 * present.
 *
 * @param element  the element to be added to this set
 */
public void add (T element)
{
  if (!(contains(element)))
  {
    LinearNode<T> node = new LinearNode<T> (element);
    node.setNext(contents);
    contents = node;
    count++;
  }
}
```

The `add` method creates a new `LinearNode<T>` object, using its constructor to store the element. Then the new node's `next` reference is set to the first node in the list, the reference to the first node is reset to point to the newly created one, and the count is incremented. Like the `add` method for the array implementation, this method consists only of simple assignment statements and is O(1).

The removeRandom Operation

The removeRandom method demonstrates how a linked list solution is sometimes more complicated than its array-based counterpart. In the ArraySet<T> class, the removeRandom method simply chose a random index into the array and returned the corresponding element. In this version of removeRandom, we must traverse the list, counting the elements until we get to the one that has been randomly selected for removal:

```java
/**
 * Removes and returns a random element from this set. Throws
 * an EmptyCollectionException if the set contains no elements.
 *
 * @return                    a random element from this set
 * @throws EmptyCollectionException if an empty set exception occurs
 */
public T removeRandom() throws EmptyCollectionException
{
  LinearNode<T> previous, current;
  T result = null;

  if (isEmpty())
    throw new EmptyCollectionException("Set");

  int choice = rand.nextInt(count) + 1;

  if (choice == 1)
  {
    result = contents.getElement();
    contents = contents.getNext();
  }
  else
  {
    previous = contents;
    for (int skip=2; skip < choice; skip++)
      previous = previous.getNext();
    current = previous.getNext();
    result = current.getElement();
    previous.setNext(current.getNext());
  }
  count--;
  return result;
}
```

Like the ArraySet<T> version, this method throws an EmptyCollectionException if there are no elements in the set. If there is at least one, a random number is chosen in the proper range. If the first element is chosen for removal, that situation is handled separately to maintain the reference to the front of the list.

Note that unlike the ArraySet<T> version, we cannot simply access the element indexed by our random number. Instead we must traverse the list resulting in a time complexity of O(n).

The remove Operation

The remove method follows somewhat similar logic to the removeRandom method, except that it is looking for a particular element to remove. If the first element matches the target element, it is removed. Otherwise, previous and current references are used to traverse the list to the appropriate point.

```
/**
 * Removes and returns the specified element from this set.
 * Throws an EmptyCollectionException if the set is empty and a
 * NoSuchElementException if the target is not in the set.
 *
 * @param target                    the element being sought in this set
 * @return                          the element just removed from this set
 * @throws EmptyCollectionException  if an empty set exception occurs
 * @throws NoSuchElementException    if a no such element exception occurs
 */
public T remove (T target) throws EmptyCollectionException,
                                  NoSuchElementException
{
  boolean found = false;
  LinearNode<T> previous, current;
  T result = null;

  if (isEmpty())
    throw new EmptyCollectionException("Set");

  if (contents.getElement().equals(target))
  {
    result = contents.getElement();
    contents = contents.getNext();
  }
  else
  {
    previous = contents;
    current = contents.getNext();
```

```
      for (int look=0; look < count && !found; look++)
        if (current.getElement().equals(target))
          found = true;
        else
        {
          previous = current;
          current = current.getNext();
        }
      if (!found)
        throw new NoSuchElementException();

        result = current.getElement();
        previous.setNext(current.getNext());
    }
    count--;

    return result;
  }
```

There is always the possibility that the target element will not be found in the collection. In that case, a `NoSuchElementException` is thrown. If the exception is not thrown, the element found is stored so that it can be returned at the end of the method. Similarly to the `removeRandom` method, this method must traverse the list to find the element to be removed resulting in a time complexity of O(n).

Other Operations

The remaining methods are similar to their counterparts in the `ArraySet<T>` class from the previous section, and are left as programming projects.

15.5 Maps and the Java Collections API

Maps are very similar to sets with the exception that the elements to be stored in the collection are separated into a key element and the associated data. Then only the key, and a reference to its associated data, are stored in the collection. This provides a couple of potential advantages. First, the data can then be part of multiple collections without duplication. Second, the collections themselves can be much smaller given that they only contain keys. In the terminology of the Java Collections API, all of the collections that we have discussed thus far could be considered sets (except that sets do not allow duplicates), because the data or element stored in each collection contains all of the data associated with that object. For example, if we were creating an ordered list of employees, ordered by name, then we would have created an `employee` object that contained all of the data for each employee, including the name and a `compareTo` method to test the name, and we would have used our operations for an ordered list to add those employees into the list. Modifying our set implementations to create map implementations is left as a programming project.

KEY CONCEPT
The difference between a set and map is that a set stores the key and the data together while a map separates the key from the data and only stores the key and a reference to the data.

However, in this same scenario, if we wanted to create an ordered list that is a map, we would have created a class to represent the name of each employee and a reference that would point to a second class that contains all of the rest of the employee data. We would have then used our ordered list operations to load the first class into our list, while the objects of the second class could exist anywhere in memory. The first class in this case is referred to as the *key*, whereas the second class is referred to as the *data*.

In this way, as we manipulate elements of the list, we are only dealing with the key, the name, and the reference, which is a much smaller segment of memory than if we were manipulating all of the data associated with an employee. We also have the advantage that the same employee data could be referenced by multiple maps without having to make multiple copies. Thus, if for one application we wanted to represent employees in a stack collection while for another application we needed to represent employees as an ordered list, we could load keys into a stack and load matching keys into an ordered list while only having one instance of the actual data. Like any situation dealing with aliases (i.e., multiple references to the same object), we must be careful that changes to an object through one reference affect the object referenced by all of the other references because there is only one instance of the object.

In Chapter 10, we discussed the Java Collections API implementations of `TreeSet` and `TreeMap`. In Chapter 14, we discussed the Java Collections API implementations of sets and maps using hashing. Note that these implementations of a set violate the algebraic notion of a set without any specific ordering to its elements. In addition to these, there are a number of other classes that implement either the `java.util.Set` or `java.util.Map` interfaces. So why, in many cases, would the creators of the Java language use sets and maps in place of the traditional data structures we have discussed in this text. Simply put, Java was not designed specifically for education. Instead, it was designed to solve industrial problems quickly and efficiently. Therefore many of the Java solutions can be described as expedient rather than elegant.

KEY CONCEPT
In the Java Collections API, sets and maps are interfaces with a wide variety of implementations.

Of course, the opposite question is equally important. If the creators of Java have chosen to use sets and maps in place of many of the traditional data structures presented in this text, why have we spent so much time talking about traditional data structures? Although there are many good answers to this question, there is one answer that supersedes all others. Although Java is a good language, it is unlikely that it is the only language you will encounter in your career. Languages have come and gone many times in the short history of our profession, but the foundational principles of data structures have remained the same. It is important that we all learn these foundational principles so that we do not have to repeat the failures of the past.

Summary of Key Concepts

- A set is a nonlinear collection in which there is essentially no inherent organization to the elements in the collection.
- The implementation of the collection operations should not affect the way users interact with the collection.
- In the Java Collections API and throughout this text, class names indicate both the underlying data structure and the collection.
- The difference between a set and map is that a set stores the key and the data together while a map separates the key from the data and only stores the key and a reference to the data.
- In the Java Collections API, sets and maps are interfaces with a wide variety of implementations.

Self-Review Questions

SR 15.1 What is a set?

SR 15.2 What would the time complexity be for the `size` operation if there were not a `count` variable?

SR 15.3 What would the time complexity be for the `add` operation if there were not a `count` variable?

SR 15.4 What do the `LinkedSet<T>` and `ArraySet<T>` classes have in common?

SR 15.5 What would be the time complexity of the `add` operation if we chose to add at the end of the list instead of the front?

SR 15.6 What is the difference between a set and a map?

SR 15.7 What are the potential advantages of a map over a set?

Exercises

EX 15.1 Define the concept of a set. List additional operations that might be considered for a set.

EX 15.2 List each occurrence of one method in the `ArraySet<T>` class calling on another method from the same class. Why is this good programming practice?

EX 15.3 Write an algorithm for the `add` method that would place each new element in position 0 of the array. What would the time complexity be for this algorithm?

EX 15.4 A bag is a very similar construct to a set except that duplicates are allowed in a bag. What changes would have to be made to our methods to create an implementation of a bag?

EX 15.5 Draw a UML diagram showing the relationships among the classes involved in the linked list implementation of a set.

EX 15.6 Write an algorithm for the `add` method that will add at the end of the list instead of the beginning. What is the time complexity of this algorithm?

EX 15.7 Modify the algorithm from the previous exercise so that it makes use of a `rear` reference. How does this affect the time complexity of this and the other operations?

EX 15.8 Discuss the effect on all the operations if there were not a `count` variable in the implementation.

EX 15.9 Discuss the impact (and draw an example) of using a dummy record at the head of the list.

Programming Projects

PP 15.1 Complete the implementation of the `ArraySet<T>` class by providing the definitions for the `size`, `isEmpty`, `iterator` and `toString` methods.

PP 15.2 Complete the implementation of the `LinkedSet<T>` class by providing the definitions for the `size`, `isEmpty`, `addAll`, `union`, `contains`, `equals`, `iterator`, and `toString` methods.

PP 15.3 Modify the `ArraySet<T>` class such that it puts the user in control of the set's capacity. Eliminate the automatic expansion of the array. The revised class should throw a `FullCollectionException` when an element is added to a full set. Add a method called `isFull` that returns true if the set is full. And add a method that the user can call to expand the capacity by a particular number of cells.

PP 15.4 An additional operation that might be implemented for a set is `difference`. This operation would take a set as a parameter and subtract the contents of that set from the current set if they exist in the current set. The result would be returned in a new set.

Implement this operation for an array implementation of a set. Be careful to consider possible exceptional situations.

PP 15.5 Implement the `difference` operation from the previous project for a linked implementation of a set.

PP 15.6 Another operation that might be implemented for a set is `intersection`. This operation would take a set as a parameter and would return a set containing those elements that exist in both sets. Implement this operation for an array implementation of a set.

PP 15.7 Implement the `intersection` operation from the previous project for a linked implementation of a set.

PP 15.8 Another operation that might be implemented for a set is `count`. This operation would take an element as a parameter and return the number of copies of that element in the set. Implement this operation for an array implementation of a set. Be careful to consider possible exceptional situations.

PP 15.9 Implement the `count` operation from the previous project for a linked implementation of a set.

PP 15.10 A bag is a very similar construct to a set except that duplicates are allowed in a bag. Implement a bag collection by creating both a `BagADT<T>` interface and an `ArrayBag<T>` class. Include the additional operations described in the earlier projects.

PP 15.11 Implement a bag collection by creating both a `BagADT<T>` interface and a `LinkedBag<T>` class. Include the additional operations described in the earlier projects.

PP 15.12 Create a new version of the `LinkedSet<T>` class that makes use of a dummy record at the head of the list.

PP 15.13 Create a simple graphical application that will allow a user to perform `add`, `remove`, and `removeRandom` operations on a set and display the resulting set (using `toString`) in a text area.

PP 15.14 Use the array implementation of a set presented in this chapter as a guide to create an array implementation of a map.

PP 15.15 Use the linked implementation of a set presented in this chapter as a guide to create a linked implementation of a map.

Answers to Self-Review Questions

SRA 15.1 A set is a collection in which there is no particular order or relationship among the elements in the collection.

SRA 15.2 Without a `count` variable, the most likely solution would be to traverse the array using a `while` loop, counting as you go, until you encounter the first null element of the array. Thus, this operation would be O(n).

SRA 15.3 Without a `count` variable, the most likely solution would be to traverse the array using a `while` loop until you encounter the first null element of the array. The new element would then be added into this position. Thus, this operation would be O(n).

SRA 15.4 Both the `LinkedSet<T>` and `ArraySet<T>` classes implement the `SetADT<T>` interface. This means that they both represent a set collection, providing the necessary operations needed to use a set. Though they both have distinct approaches to managing the collection, they are functionally interchangeable from the user's point of view.

SRA 15.5 To add at the end of the list, we would have to traverse the list to reach the last element. This traversal would cause the time complexity to be O(n). An alternative would be to modify the solution to add a `rear` reference that always pointed to the last element in the list. This would help the time complexity for `add` but would have consequences if we try to remove the last element.

SRA 15.6 A set contains all of the data associated with the elements that are stored within it whereas a map contains only a key and a reference to the rest of the data that is stored elsewhere.

SRA 15.7 First, the data can then be part of multiple collections without duplication. Second, the collections themselves can be much smaller given that they only contain keys.

This page intentionally left blank

Appendix

A UML

The Unified Modeling Language (UML)

Software engineering deals with the analysis, synthesis, and communication of ideas in the development of software systems. In order to facilitate the methods and practices necessary to accomplish these goals, software engineers have developed a wide variety of notations to capture and communicate information. Although numerous notations are available, only a few have become popular, one of which in particular has become a de facto standard in the industry.

KEY CONCEPT
The Unified Modeling Language (UML) provides a notation with which we can capture and illustrate program designs.

The *Unified Modeling Language* (UML) was developed in the mid-1990s, but is actually the synthesis of three separate and long-standing design notations, each popular in its own right. We use UML notation throughout this book to illustrate program designs, and this section describes the key aspects of UML diagrams. Keep in mind that UML is language-independent. It uses generic terms and contains some features that are not relevant to the Java programming language. We focus on aspects of UML that are particularly appropriate for its use in this book.

UML is an object-oriented modeling language. It provides a convenient way to represent the relationships among classes and objects in a software system. We provide an overview of UML here, and use it throughout the book. The details of the underlying object-oriented concepts are discussed in Appendix B.

UML Class Diagrams

A UML *class diagram* describes the classes in the system, the static relationships among them, the attributes and operations associated with a class, and the constraints on the connections among objects. The terms “attribute” and “operation” are generic object-oriented terms. An *attribute* is any class-level data including variables and constants. An *operation* is essentially equivalent to a method.

A class is represented in UML by a rectangle, usually divided into three sections containing the class name, its attributes, and its operations. Figure A.1

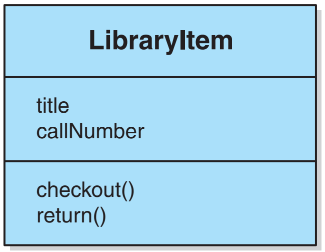


FIGURE A.1 `LibraryItem` class diagram

illustrates a class named `LibraryItem`. There are two attributes associated with the class, `title` and `callNumber`, and there are two operations associated with the class, `checkout` and `return`.

In the notation for a class, the attributes and operations are optional. Therefore, a class may be represented by a single rectangle containing only the class name, if desired. We can include the attributes and/or operations whenever they help convey important information in the diagram. If attributes or operations are included, then both sections are shown (though not necessarily filled) to make it clear which is which.

There are many additional pieces of information that can be included in the UML class notation. An annotation bracketed using < and > is called a *stereotype* in UML terminology. The `<abstract>` stereotype or the `<interface>` stereotype could be added above the name to indicate that it is representing an abstract class or an interface. The visibility of a class is assumed to be public by default, though nonpublic classes can be identified using a property string in curly braces, such as `{private}`.

Attributes listed in a class can also provide several pieces of additional information. The full syntax for showing an attribute is

visibility name : *type* = *default-value*

The visibility may be spelled out as `public`, `protected`, or `private`, or you may use the symbols + to represent public visibility, # for protected visibility, or – for private visibility. For example, we might have listed the title of a `LibraryItem` as

```
- title : String
```

indicating that the attribute `title` is a private variable of type `String`. A default value is not provided in this case. Also, the stereotype `<final>` may be added to an attribute to indicate that it is a constant.

Similarly, the full syntax for an operation is

visibility name (*parameter-list*) : *return-type* {*property-string*}

As with the syntax for attributes, all of the items other than the name are optional. The visibility modifiers are the same as they are for attributes. The *parameter-list* can include the name and type of each parameter, separated by a colon. The *return-type* is the type of the value returned from the operation.

KEY CONCEPT
Various kinds of relationships can be represented in a UML class diagram.

UML Relationships

There are several kinds of relationships among classes that UML diagrams can represent. Usually they are shown as lines or arrows connecting one class to

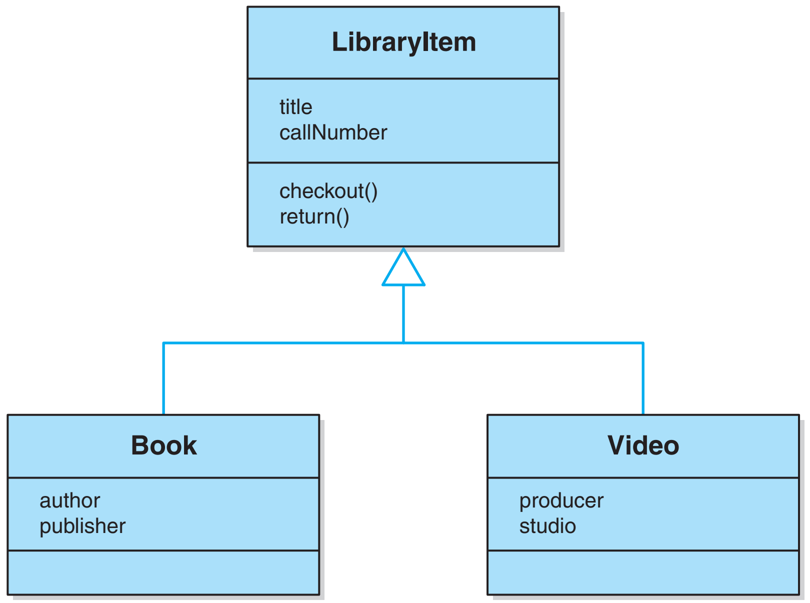


FIGURE A.2 A UML class diagram showing inheritance relationships

another. Specific types of lines and arrowheads have specific meaning in UML.

KEY CONCEPT
The inheritance relationship is indicative of one class being derived from or being a child of the other class.

One type of relationship shown between two classes in a UML diagram is an *inheritance relationship*. Figure A.2 shows two classes that are derived from the `LibraryItem` class. Inheritance is shown using an arrow with an open arrowhead pointing from the child class to the parent class. This example shows that both the `Book` class and the `Video` class inherit all of the attributes and operations of `LibraryItem`, but they also extend that definition with attributes of their own. Note that in this example, neither subclass has any additional operations other than those provided in the parent class.

KEY CONCEPT
The association relationship represents relationships between instances of classes.

Another relationship shown in a UML diagram is an *association*, which represents relationships between instances (objects) of the classes. An association is indicated by a solid line between the two classes involved and can be annotated with the *cardinality* of the relationship on either side. For example, Figure A.3 shows an association between a `LibraryCustomer` and a `LibraryItem`. The cardinality of 0..* means "zero or more," in this case indicating that any given library customer may check out 0 or more items, and that any given library item may be

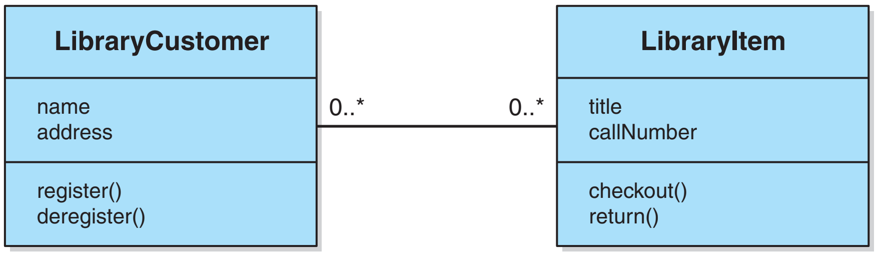


FIGURE A.3 A UML class diagram showing an association

checked out by multiple customers. The cardinality of an association may indicate other relationships, such as an exact number or a specific range. For example, if a customer is allowed to check out no more than five items, the cardinality could have been indicated by 0..5.

KEY CONCEPT

The aggregation relationship represents one class being made up of other classes.

A third type of relationship between classes is the concept of *aggregation*. This is the situation in which one class is essentially made up, at least in part, of other classes. For example, we can extend our library example to show a `CourseMaterials` class that is made up of books, course notes, and videos, as shown in Figure A.4. Aggregation is shown by using an open diamond on the aggregate end of the relationship.

KEY CONCEPT

The implementation relationship represents a class implementing an interface.

A fourth type of relationship that we may wish to represent is the concept of *implementation*. This relationship occurs between an interface and any class that implements that interface. Figure A.5 shows an interface called `Copyrighted` that contains two abstract

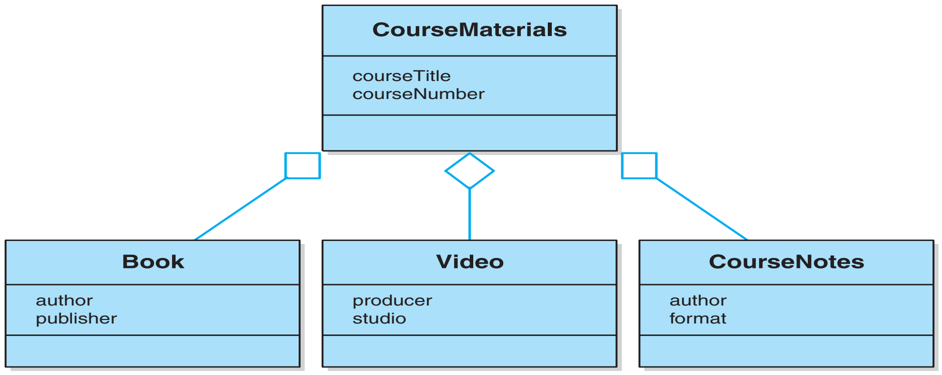


FIGURE A.4 One class shown as an aggregate of other classes

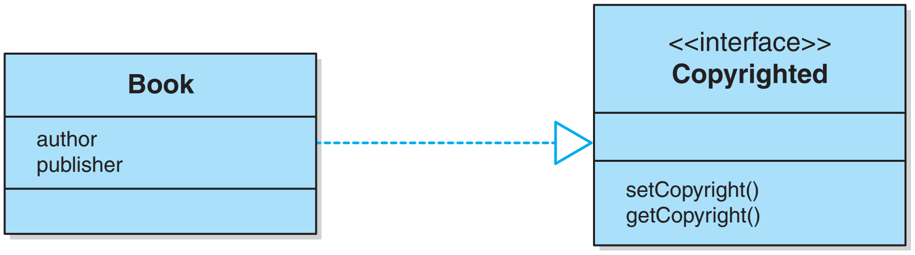


FIGURE A.5 A UML diagram showing a class implementing an interface

KEY CONCEPT

The uses relationship represents one class using another.

methods. The dotted arrow with the open arrowhead indicates that the `Book` class implements the `Copyrighted` interface.

A fifth type of relationship between classes is the concept of one class *using* another. Examples of this concept include such things as an instructor using a chalkboard, a driver using a car, or a library customer using a computer. Figure A.6 illustrates this relationship, showing that a `LibraryCustomer` might use a `Computer`. The uses relationship is indicated by a dotted line with an open arrowhead that is usually annotated with the nature of the relationship.

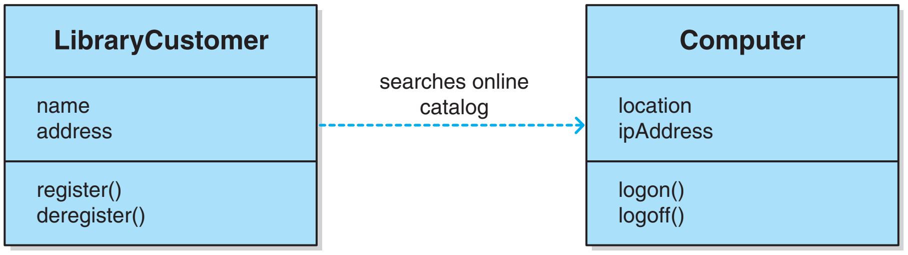


FIGURE A.6 One class indicating its use of another

Summary of Key Concepts

- The Unified Modeling Language (UML) provides a notation with which we can capture and illustrate program designs.
- Various kinds of relationships can be represented in a UML class diagram.
- The inheritance relationship is indicative of one class being derived from or being a child of the other class.
- The association relationship represents relationships between instances of classes.
- The aggregation relationship represents one class being made up of other classes.
- The implementation relationship represents a class implementing an interface.
- The uses relationship represents one class using another.

Self-Review Questions

SR A.1 What does a UML class diagram represent?

SR A.2 What are the different types of relationships represented in a class diagram?

Exercises

EX A.1 Create a UML class diagram for the organization of a university, where the university is made up of colleges, which are made up of departments, which contain faculty and students.

EX A.2 Complete the UML class description for a library system outlined in this chapter.

EX A.3 List and illustrate an example of each of the relationships discussed in this chapter.

Answers to Self-Review Questions

SRA A.1 A class diagram describes the types of objects or classes in the system, the static relationships among them, the attributes and operations of a class, and the constraints on the connections among objects.

SRA A.2 Relationships shown in a UML class diagram include subtypes or extensions, associations, aggregates, and the implementation of interfaces.

This page intentionally left blank

Appendix B

Object-Oriented Design

B.1 Overview of Object-Orientation

Java is an object-oriented language. As the name implies, an *object* is a fundamental entity in a Java program. In addition to objects, a Java program also manages primitive data. *Primitive data* includes common, fundamental values such as numbers and characters. An object usually represents something more specialized or complex, such as a bank account. An object often contains primitive values and is in part defined by them. For example, an object that represents a bank account might contain the account balance, which is stored as a primitive numeric value.

An object is defined by a *class*, which can be thought of as the data type of the object. The operations that can be performed on the object are defined by the methods in the class.

Once a class has been defined, multiple objects can be created from that class. For example, once we define a class to represent the concept of a bank account, we can create multiple objects that represent specific, individual bank accounts. Each bank account object would keep track of its own balance. This is an example of *encapsulation*, meaning that each object protects and manages its own information. The methods defined in the bank account class would allow us to perform operations on individual bank account objects. For instance, we might withdraw money from a particular account. We can think of these operations as services that the object performs. The act of invoking a method on an object is sometimes referred to as *sending a message* to the object, requesting that the service be performed.

Classes can be created from other classes using *inheritance*. That is, the definition of one class can be based on another class that already exists. Inheritance is a form of software *reuse*, capitalizing on the similarities between various kinds of classes that we may want to create. One class can be used to derive several new classes. Derived classes can then be used to derive even more classes. This creates a hierarchy of classes, where characteristics defined in one class are inherited by its children, which in turn pass them on to their children, and so on. For example, we might create a hierarchy of classes that represent various types of accounts. Common characteristics are defined in high-level classes, and specific differences are defined in derived classes.

Classes, objects, encapsulation, and inheritance are the primary ideas that make up the world of object-oriented software. They are depicted in Figure B.1 and are explored in more detail throughout this appendix.

B.2 Using Objects

The following `println` statement illustrates the process of using an object for the services it provides:

```
System.out.println ("Whatever you are, be a good one.");
```

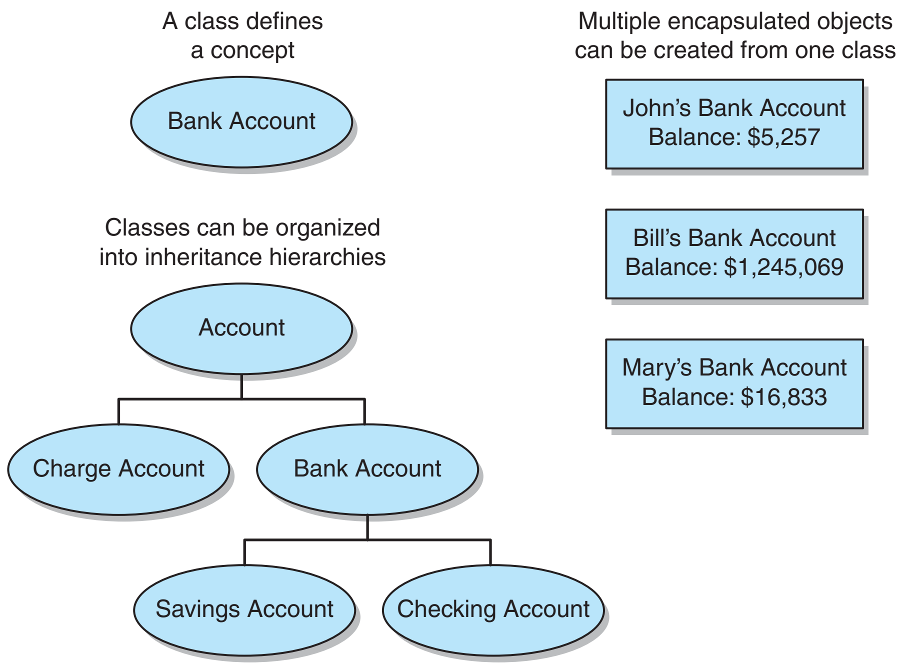

A class defines a concept
Bank Account
Classes can be organized into inheritance hierarchies
Account
Charge Account
Bank Account
Savings Account
Checking Account
Multiple encapsulated objects can be created from one class
John's Bank Account Balance: $5,257
Bill's Bank Account Balance: $1,245,069
Mary's Bank Account Balance: $16,833

FIGURE B.1 Various aspects of object-oriented software

The `System.out` object represents an output device or file, which by default is the monitor screen. To be more precise, the object's name is `out`, and it is stored in the `System` class.

The `println` method represents a service that the `System.out` object performs for us. Whenever we request it, the object will print a string of characters to the screen. We can say that we send the `println` message to the `System.out` object to request that some text be printed.

Abstraction

An object is an *abstraction*, meaning that the precise details of how it works are irrelevant from the point of view of the user of the object. We don't really need to know how the `println` method prints characters to the screen, as long as we can count on it to do its job. Of course, there are times when it is helpful to understand such information, but it is not necessary in order to *use* the object.

Sometimes it is important to hide or ignore certain details. A human being is capable of mentally managing around seven (plus or minus two) pieces of information in short-term memory. Beyond that, we start to lose track of some of the pieces. However, if we group pieces of information together, then those pieces can be managed as one "chunk" in our minds. We don't actively deal with all of the details in the chunk, but we can still manage it as a single entity. Therefore, we can deal with large quantities of information by organizing it into chunks. An object is a construct that organizes information and allows us to hide the details inside. An object is therefore a wonderful abstraction.

We use abstractions every day. Think about a car for a moment. You don't necessarily need to know how a four-cycle combustion engine works in order to drive a car. You just need to know some basic operations: how to turn it on, how to put it in gear, how to make it move with the pedals and steering wheel, and how to stop it. These operations define the way a person interacts with the car. They mask the details of what is happening inside the car that allow it to function. When you are driving a car, you are not usually thinking about the spark plugs igniting the gasoline that drives the piston that turns the crankshaft that turns the axle that turns the wheels. If we had to worry about all of these underlying details, we would never be able to operate something as complicated as a car.

Initially, all cars had manual transmissions. The driver had to understand and deal with the details of changing gears with the stick shift. Eventually, automatic transmissions were developed, and the driver no longer had to worry about shifting gears. Those details were hidden by raising the *level of abstraction*.

Of course, someone has to deal with the details. The car manufacturer has to know the details in order to design and build the car in the first place. A car mechanic relies on the fact that most people don't have the expertise or tools necessary to fix a car when it breaks.

KEY CONCEPT

An abstraction hides details. A good abstraction hides the right details at the right time so that we can manage complexity.

The level of abstraction must be appropriate for each situation. Some people prefer to drive a manual transmission car. A race car driver, for instance, needs to control the shifting manually for optimum performance.

Likewise, someone has to create the code for the objects we use. Later in this chapter, we explore how to define objects by creating classes. For now, we can create and use objects from classes that have been defined for us already. Abstraction makes that possible.

Creating Objects

A Java variable can hold either a primitive value or a *reference to an object*. Like variables that hold primitive types, a variable that serves as an object reference must be declared. A class is used to define an object, and the class name can be thought of as the type of an object. The declarations of object references have a similar structure to the declarations of primitive variables.

The following declaration creates a reference to a `String` object:

```
String name;
```

That declaration is like the declaration of an integer, in that the type is followed by the variable name we want to use. However, no `String` object actually exists yet. To create an object, we use the `new` operator:

```
name = new String ("James Gosling");
```

The act of creating an object by using the `new` operator is called *instantiation*. An object is said to be an *instance* of a particular class. After the `new` operator creates the object, a *constructor* is invoked to help set it up initially. A constructor has the same name as the class, and is similar to a method. In this example, the parameter to the constructor is a string literal that specifies the characters that the `String` object will hold.

KEY CONCEPT

The `new` operator returns a reference to a newly created object.

The acts of declaring the object reference variable and creating the object itself can be combined into one step by initializing the variable in the declaration, just as we do with primitive types:

```
String name = new String ("James Gosling");
```

After an object has been instantiated, we use the *dot operator* to access its methods. The dot operator is appended directly after the object reference, followed by the method being invoked. For example, to invoke the `length` method defined in the `String` class, we use the dot operator on the `name` reference variable:

```
count = name.length();
```

An object reference variable (such as `name`) actually stores the address where the object is stored in memory. However, we don't usually care about the actual address value. We just want to access the object, wherever it is.

Even though they are not primitive types, strings are so fundamental and frequently used that Java defines string literals delimited by double quotation marks, as we have seen in various examples. This is a shortcut notation. Whenever a string literal appears, a `String` object is created. Therefore, the following declaration is valid:

```
String name = "James Gosling";
```

That is, for `String` objects, the explicit use of the `new` operator, and the call to the constructor can be eliminated. In most cases, this simplified syntax for strings is used.

B.3 Class Libraries and Packages

A *class library* is a set of classes that supports the development of programs. A compiler often comes with a class library. Class libraries can also be obtained separately through third-party vendors. The classes in a class library contain methods that are often valuable to a programmer because of the special functionality they offer. In fact, programmers often become dependent on the methods in a class library and begin to think of them as part of the language. But, technically, they are not in the language definition.

The `String` class, for instance, is not an inherent part of the Java language. It is part of the Java *standard class library* that can be found in any Java development environment. The classes that make up the library were created by employees at Sun Microsystems, the company that created the Java language.

KEY CONCEPT
The Java standard class library is a useful set of classes that anyone can use when writing Java programs.

The class library is made up of several clusters of related classes, which are sometimes called Java APIs. API stands for *application programmer interface*. For example, we may refer to the Java Database API when we are talking about the set of classes that helps us to write programs that interact with a database. Another example of an API is the Java Swing API, which refers to a set of classes that defines special graphical components used in a graphical user interface. Sometimes the entire standard library is referred to generically as the Java API.

KEY CONCEPT
A package is a Java language element used to group related classes under a common name.

The classes of the Java standard class library are also grouped into *packages*, which, like the APIs, let us group related classes by one name. Each class is part of a particular package. The `String` class and the `System` class, for example, are both part of the `java.lang` package.

The package organization is more fundamental and language-based than the API names. Though there is a general correspondence between package and API names, the groups of classes that make up a given API might cross packages. We primarily refer to classes in terms of their package organization in this text.

The `import` Declaration

The classes of the package `java.lang` are automatically available for use when writing a program. To use classes from any other package, however, we must either *fully qualify* the reference or use an `import` *declaration*.

When you want to use a class from a class library in a program, you could use its fully qualified name, including the package name, every time it is referenced. For example, every time you want to refer to the `Random` class that is defined in the `java.util` package, you can write `java.util.Random`. However, completely specifying the package and class name every time it is needed quickly becomes tiring. Java provides the `import` declaration to simplify these references.

The `import` declaration identifies the packages and classes that will be used in a program, so that the fully qualified name is not necessary with each reference. The following is an example of an `import` declaration:

```
import java.util.Random;
```

This declaration asserts that the `Random` class of the `java.util` package may be used in the program. Once this `import` declaration is made, it is sufficient to use the simple name `Random` when referring to that class in the program.

Another form of the `import` declaration uses an asterisk (*) to indicate that any class inside the package might be used in the program. Therefore, the declaration

```
import java.util.*;
```

allows all classes in the `java.util` package to be referenced in the program without the explicit package name. If only one class of a particular package will be used in a program, it is usually better to name the class specifically in the `import` declaration. However, if two or more classes will be used, the `*` notation is fine. Once a class is imported, it is as if its code has been brought into the program. The code is not actually moved, but that is the effect.

The classes of the `java.lang` package are automatically imported because they are fundamental and can be thought of as basic extensions to the language. Therefore, any class in the `java.lang` package, such as `String`, can be used without an explicit `import` declaration. It is as if all programs automatically contain the following declaration:

```
import java.lang.*;
```

B.4 State and Behavior

Think about objects in the world around you. How would you describe them? Let's use a ball as an example. A ball has particular characteristics such as its diameter, color, and elasticity. Formally, we say the properties that describe an object, called *attributes*, define the object's *state of being*. We also describe a ball by what it does, such as the fact that it can be thrown, bounced, or rolled. These activities define the object's *behavior*.

All objects have a state and a set of behaviors. We can represent these characteristics in software objects as well. The values of an object's variables describe the object's state, and the methods that can be invoked using the object define the object's behaviors.

KEY CONCEPT
Each object has a state and a set of behaviors. The values of an object's variables define its state. The methods to which an object responds define its behaviors.

Consider a computer game that uses a ball. The ball could be represented as an object. It could have variables to store its size and location, and methods that draw it on the screen and calculate how it moves when thrown, bounced, or rolled. The variables and methods defined in the ball object establish the state and behavior that are relevant to the ball's use in the computerized ball game.

Each object has its own state. Each ball object has a particular location, for instance, which typically is different from the location of all other balls. Behaviors, though, tend to apply to all objects of a particular type. For instance, in general, any ball can be thrown, bounced, or rolled. The act of rolling a ball is generally the same for all balls.

The state of an object and that object's behaviors work together. How high a ball bounces depends on its elasticity. The action is the same, but the specific result depends on that particular object's state. An object's behavior often modifies its state. For example, when a ball is rolled, its location changes.

Any object can be described in terms of its state and behavior. Let's consider another example. In software that is used to manage a university, a student could be represented as an object. The collection of all such objects represents the entire student body at the university. Each student has a state. That is, each student object contains the variables that store information about a particular student, such as name, address, major, courses taken, grades, and grade point average. A student object also has behaviors. For example, the class of the student object may contain a method to add a new course.

Although software objects often represent tangible items, they don't have to. For example, an error message can be an object, with its state being the text of the message, and behaviors including the process of issuing (perhaps printing) the error. A common mistake made by new programmers to the world of object-orientation is to limit the possibilities to tangible entities.

B.5 Classes

An object is defined by a class. A class is the model, pattern, or blueprint from which an object is created. Consider the blueprint created by an architect when designing a house. The blueprint defines the important characteristics of the house: walls, windows, doors, electrical outlets, and so forth. Once the blueprint is created, several houses can be built using it.

In one sense, the houses built from the blueprint are different. They are in different locations, have different addresses, contain different furniture, and different people live in them. Yet, in many ways they are the "same" house. The layout of the rooms and other crucial characteristics are the same in each. To create a different house, we would need a different blueprint.

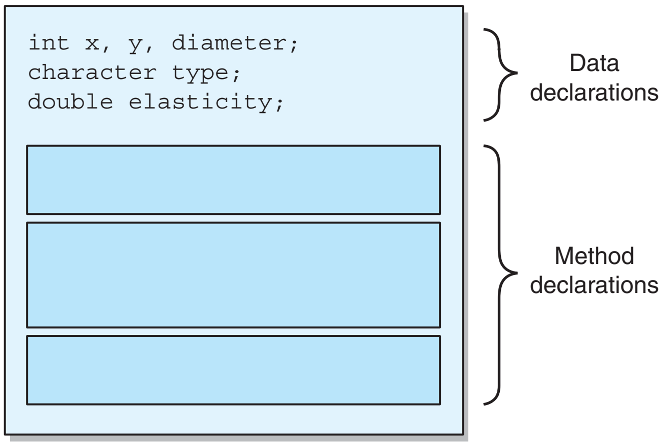


FIGURE B.2 The members of a class: data and method declarations

A class is a blueprint of an object. But a class is not an object any more than a blueprint is a house. In general, no space to store data values is reserved in a class. To allocate space to store data values, we have to instantiate one or more objects from the class (static data is the exception to this rule and is discussed later in this chapter). Each object is an instance of a class. Each object has space for its own data, which is why each object can have its own state.

KEY CONCEPT

A class is a blueprint for an object; it reserves no memory space for data. Each object has its own data space, and thus its own state.

A class contains the declarations of the data that will be stored in each instantiated object, and the declarations of the methods that can be invoked using an object. Collectively these are called the *members* of the class. See Figure B.2.

Consider the class shown in Listing B.1, called `Coin`, that represents a coin that can be flipped and that at any point in time shows a face of either heads or tails.

In the `Coin` class, we have two integer constants, `HEADS` and `TAILS`, and one integer variable, `face`. The rest of the `Coin` class is composed of the `Coin` constructor and three regular methods: `flip`, `isHeads`, and `toString`.

Constructors are special methods that have the same name as the class. The `Coin` constructor gets called when the `new` operator is used to create a new instance of the `Coin` class. The rest of the methods in the `Coin` class define the various services provided by `Coin` objects.

A class we define can be used in multiple programs. This is no different from using the `String` class in whatever program we need it. When designing a class, it is always good to look to the future to try to give the class behaviors that may be beneficial in other programs, not just fit the specific purpose for which you are creating it at the moment.

LISTING B.1

```
/**
 * Coin represents a coin with two sides that can be flipped.
 *
 * @author Lewis
 * @author Chase
 * @version 1.0, 8/18/08
 */

public class Coin
{
   private final int HEADS = 0;
   private final int TAILS = 1;

   private int face;


   /**
    * Sets up the coin by flipping it initially.
    */
   public Coin ()
   {
      flip();
   }

   /**
    * Flips the coin by randomly choosing a face value.
    */

   public void flip ()
   {
      face = (int) (Math.random() * 2);
   }

   /**
    * Returns true if the current face of the coin is heads.
    *
    * @return true if the face is heads
    */

   public boolean isHeads ()
   {
      return (face == HEADS);
   }
```

LISTING B.1 *continued*

```
   /**
    * Returns the current face of the coin as a string.
    *
    * @return the string representation of the current face value of this coin
    */

   public String toString()
   {
      String faceName;

      if (face == HEADS)
         faceName = "Heads";
      else
         faceName = "Tails";

      return faceName;
   }
}
```

Instance Data

KEY CONCEPT

The scope of a variable, which determines where it can be referenced, depends on where it is declared.

Note that in the `Coin` class, the constants `HEADS` and `TAILS` and the variable `face` are declared inside the class, but not inside any method. The location at which a variable is declared defines its *scope*, which is the area within a program in which that variable can be referenced. By being declared at the class level (not within a method), these variables and constants can be referenced in any method of the class.

Attributes declared at the class level are also called *instance data*, because memory space for the data is reserved for each instance of the class that is created. Each `Coin` object, for example, has its own `face` variable with its own data space. Therefore, at any point in time two `Coin` objects can have their own states: one can be showing heads and the other can be showing tails, perhaps.

Java automatically initializes any variables declared at the class level. For example, all variables of numeric types such as `int` and `double` are initialized to zero. However, despite the fact that the language performs this automatic initialization, it is good practice to initialize variables explicitly (usually in a constructor) so that anyone reading the code will clearly understand the intent.

B.6 Encapsulation

We can think about an object in one of two ways. The view we take depends on what we are trying to accomplish at the moment. First, when we are designing and implementing an object, we need to think about the details of how an object works. That is, we have to design the class; we have to define the variables that will be held in the object and define the methods that make the object useful.

However, when we are designing a solution to a larger problem, we have to think in terms of how the objects in the program interact. At that level, we have to think only about the services that an object provides, not about the details of how those services are provided. As we discussed earlier in this chapter, an object provides a level of abstraction that allows us to focus on the larger picture when we need to.

KEY CONCEPT

Objects should be encapsulated. The rest of a program should interact with an object only through a well-defined interface.

This abstraction works only if we are careful to respect its boundaries. An object should be *self-governing*, which means that the variables contained in an object should be modified only within the object. Only the methods within an object should have access to the variables in that object. We should make it difficult, if not impossible, for code outside of a class to "reach in" and change the value of a variable that is declared inside the class.

The object-oriented term for this characteristic is *encapsulation*. An object should be encapsulated from the rest of the system. It should interact with other parts of a program only through the specific set of methods that define the services provided by that object. These methods define the *interface* between that object and the program that uses it.

The code that uses an object, sometimes called the *client* of an object, should not be allowed to access variables directly. The client should interact with the object's methods, which in turn interact on behalf of the client with the data encapsulated within the object.

Visibility Modifiers

In Java, we accomplish object encapsulation using *modifiers*. A modifier is a Java reserved word that is used to specify particular characteristics of a programming language construct. For example, the `final` modifier is used to declare a constant. Java has several modifiers that can be used in various ways. Some modifiers can be used together, but some combinations are invalid.

Some Java modifiers are called *visibility modifiers* because they control access to the members of a class. The reserved words `public` and `private` are visibility modifiers that can be applied to the variables and methods of a class. If a member of a class has *public visibility*, then it can be directly referenced from outside of the object. If a member of a class has *private visibility*, it can be used anywhere inside the class definition but cannot be referenced externally. A third visibility modifier, `protected`, is relevant only in the context of inheritance, which is discussed later in this chapter.

Public variables violate encapsulation. They allow code external to the class in which the data is defined to reach in and access or modify the value of the data. Therefore, instance data should be defined with private visibility. Data that is declared as private can be accessed only by the methods of the class, which makes the objects created from that class self-governing.

Which visibility we apply to a method depends on the purpose of that method. Methods that provide services to the client of the class must be declared with public visibility so that they can be invoked by the client. These methods are sometimes referred to as *service methods*. A private method cannot be invoked from outside the class. The only purpose of a private method is to help the other methods of the class do their job. Therefore, private methods are sometimes referred to as *support methods*.

KEY CONCEPT

Instance variables should be declared with private visibility to promote encapsulation.

The table in Figure B.3 summarizes the effects of public and private visibility on both variables and methods.

Note that a client can still access or modify `private` data by invoking service methods that change the data. A class must provide service methods for valid client operations. The code of those methods must be carefully designed to permit only appropriate access and valid changes.

Giving constants public visibility is generally considered acceptable. Although their values can be accessed directly, they cannot be changed because they were declared using the `final` modifier. Keep in mind that encapsulation means that data values should not be able to be *changed* directly by another part of the code. Because constants, by definition, cannot be changed, the encapsulation issue is largely moot.

	`public`	`private`
Variables	**Violate encapsulation**	Enforce encapsulation
Methods	Provide services to clients	Support other methods in the class

FIGURE B.3 The effects of public and private visibility

UML diagrams reflect the visibility of a class member with special notations. A member with public visibility is preceded by a plus sign (+), and a member with private visibility is preceded by a minus sign (−).

Local Data

As we defined earlier, the scope of a variable or constant is the part of a program in which a valid reference to that variable can be made. A variable can be declared inside a method, making it *local data* as opposed to instance data. Recall that instance data is declared in a class but not inside any particular method. Local data has scope limited to only the method in which it is declared. Any reference to local data of one method in any other method would cause the compiler to issue an error message. A local variable simply does not exist outside of the method in which it is declared. Instance data, declared at the class level, has a scope of the entire class. Any method of the class can refer to it.

KEY CONCEPT
A variable declared in a method is local to that method and cannot be used outside of it.

Because local data and instance data operate at different levels of scope, it's possible to declare a local variable inside a method by using the same name as an instance variable declared at the class level. Referring to that name in the method will reference the local version of the variable. This naming practice obviously has the potential to confuse anyone reading the code, so it should be avoided.

The formal parameter names in a method header serve as local data for that method. They don't exist until the method is called, and cease to exist when the method is exited.

B.7 Constructors

A constructor is similar to a method that is invoked when an object is instantiated. When we define a class, we usually define a constructor to help us set up the class. In particular, we often use a constructor to initialize the variables associated with each object.

A constructor differs from a regular method in two ways. First, the name of a constructor is the same name as the class. Therefore, the name of the constructor in the `Coin` class is `Coin`, and the name of the constructor in the `Account` class is `Account`. Second, a constructor cannot return a value and does not have a return type specified in the method header.

KEY CONCEPT
A constructor cannot have any return type, even `void`.

A common mistake made by programmers is to put a `void` return type on a constructor. As far as the compiler is concerned, putting any return type on a constructor, even `void`, turns it into a regular method that happens to have the same name as the class. As such, it cannot be invoked as a constructor. This leads to error messages that are sometimes difficult to decipher.

A constructor is generally used to initialize the newly instantiated object. We don't have to define a constructor for every class. Each class has a *default constructor* that takes no parameters and is used if we don't provide our own. This default constructor generally has no effect on the newly created object.

B.8 Method Overloading

When a method is invoked, the flow of control transfers to the code that defines the method. After the method has been executed, control returns to the location of the call and processing continues.

Often the method name is sufficient to indicate which method is being called by a specific invocation. But in Java, as in other object-oriented languages, you can use the same method name with different parameter lists for multiple methods. This technique is called *method overloading*. It is useful when you need to perform similar methods on different types of data.

The compiler must still be able to associate each invocation to a specific method declaration. If the method name for two or more methods is the same, then additional information is used to uniquely identify the version that is being invoked. In Java, a method name can be used for multiple methods as long as the number of parameters, the types of those parameters, or the order of the types of parameters is distinct. A method's name along with the number, type, and order of its parameters is called the method's *signature*. The compiler uses the complete method signature to *bind* a method invocation to the appropriate definition.

KEY CONCEPT

The versions of an overloaded method are distinguished by their signatures. The number, type, or order of their parameters must be distinct.

The compiler must be able to examine a method invocation, including the parameter list, to determine which specific method is being invoked. If you attempt to specify two method names with the same signature, the compiler will issue an appropriate error message and will not create an executable program. There can be no ambiguity.

Note that the return type of a method is not part of the method signature. That is, two overloaded methods cannot differ only by their return type. The reason is that the value returned by a method can be ignored by the invocation. The compiler would not be able to distinguish which version of an overloaded method is being referenced in such situations.

The `println` method is an example of a method that is overloaded several times, each accepting a single type. Here is a partial list of its various signatures:

```
>println (String s)
>println (int i)
>println (double d)
>println (char c)
>println (boolean b)
```

The following two lines of code actually invoke different methods that have the same name:

```
System.out.println ("The total is: ");
System.out.println (count);
```

The first line invokes the `println` that accepts a string, and the second line, assuming `count` is an integer variable, invokes the version of `println` that accepts an integer. We often use a `println` statement that prints several distinct types, such as:

```
System.out.println ("The total is: " + count);
```

In this case, the plus sign is the string concatenation operator. First, the value in the variable `count` is converted to a string representation. Then the two strings are concatenated into one longer string, and the definition of `println` that accepts a single string is invoked.

Constructors are primary candidates for overloading. By providing multiple versions of a constructor, we provide several ways to set up an object.

B.9 References Revisited

In previous examples, we have declared *object reference variables* through which we access particular objects. Let's examine this relationship in more detail.

KEY CONCEPT
An object reference variable stores the address of an object.

An object reference variable and an object are two separate things. Remember that the declaration of the reference variable and the creation of the object that it refers to are separate steps. We often declare the reference variable and create an object for it to refer to on the same line, but keep in mind that we don't have to do so. In fact, in many cases, we won't want to.

The reference variable holds the address of an object even though the address never is disclosed to us. When we use the dot operator to invoke an object's method, we are actually using the address in the reference variable to locate the representation of the object in memory, look up the appropriate method, and invoke it.

The null Reference

A reference variable that does not currently point to an object is called a *null reference*. When a reference variable is initially declared as an instance variable, it is a null reference. If we try to follow a null reference, a `NullPointerException` is thrown, indicating that there is no object to reference. For example, consider the following situation:

```
class NameIsNull
{
   String name; // not initialized, therefore null

   void printName()
   {
     System.out.println (name.length()); // causes an exception
   }
}
```

The declaration of the instance variable `name` asserts it to be a reference to a `String` object, but it doesn't create any `String` object for it to refer to. The variable `name`, therefore, contains a null reference. When the method attempts to invoke the `length` method of the object to which `name` refers, an exception is thrown because no object exists to execute the method.

Note that this situation can arise only in the case of instance variables. Suppose, for instance, the following two lines of code were in a method:

```
String name;
System.out.println (name.length());
```

In this case, the variable `name` is local to whatever method it is declared in. The compiler would complain that we were using the `name` variable before it had been initialized. In the case of instance variables, however, the compiler can't determine whether a variable had been initialized or not. Therefore, the danger of attempting to follow a null reference is a problem.

The identifier `null` is a reserved word in Java and represents a null reference. We can explicitly set a reference to `null` to ensure that it doesn't point to any object. We can also use it to check whether a particular reference currently points to an object. For example, we could have used the following code in the `printName` method to keep us from following a null reference:

KEY CONCEPT

The reserved word `null` represents a reference that does not point to a valid object.

```
if (name == null)
    System.out.println ("Invalid Name");
else
    System.out.println (name.length());
```

The `this` Reference

Another special reference for Java objects is called the `this` reference. The word `this` is a reserved word in Java. It allows an object to refer to itself. As we have discussed, a method is always invoked through a particular object or class. Inside that method, the `this` reference can be used to refer to the currently executing object.

KEY CONCEPT

The `this` reference always refers to the currently executing object.

For example, in the `ChessPiece` class, there could be a method called `move`, which could contain the following line:

```
if (this.position == piece2.position)
   result = false;
```

In this situation, the `this` reference is being used to clarify which position is being referenced. The `this` reference refers to the object through which the method was invoked. So when the following line is used to invoke the method, the `this` reference refers to `bishop1`:

```
bishop1.move();
```

But when another object is used to invoke the method, the `this` reference refers to it. Therefore, when the following invocation is used, the `this` reference in the `move` method refers to `bishop2`:

```
bishop2.move();
```

The `this` reference can also be used to distinguish the parameters of a constructor from their corresponding instance variables with the same names. For example, the constructor of a class called `Account` could be defined as follows:

```
public Account (String owner, long account, double initial)
{
   name = owner;
   acctNumber = account;
   balance = initial;
}
```

In this constructor, we deliberately came up with different names for the parameters to distinguish them from the instance variables `name`, `acctNumber`, and `balance`. This distinction is arbitrary. The constructor could have been written as follows using the `this` reference:

```
public Account (String name, long acctNumber, double balance)
{
   this.name = name;
   this.acctNumber = acctNumber;
   this.balance = balance;
}
```

In this version of the constructor, the `this` reference specifically refers to the instance variables of the object. The variables on the right-hand side of the assignment statements refer to the formal parameters. This approach eliminates the need to come up with different yet equivalent names. This situation sometimes occurs in other methods, but comes up often in constructors.

Aliases

Because an object reference variable stores an address, programmers must be careful when managing objects. In particular, the semantics of an assignment statement for objects must be carefully understood. First, let's review the concept of assignment for primitive types. Consider the following declarations of primitive data:

```
int num1 = 5;
int num2 = 12;
```

In the following assignment statement, a copy of the value that is stored in num1 is stored in num2:

```
num2 = num1;
```

The original value of 12 in num2 is overwritten by the value 5. The variables num1 and num2 still refer to different locations in memory, and both of those locations now contain the value 5.

Now consider the following object declarations:

```
ChessPiece bishop1 = new ChessPiece();
ChessPiece bishop2 = new ChessPiece();
```

Initially, the references bishop1 and bishop2 refer to two different ChessPiece objects. The following assignment statement copies the value in bishop1 into bishop2:

```
bishop2 = bishop1;
```

The key issue is that when an assignment like this is made, the address stored in bishop1 is copied into bishop2. Originally the two references referred to different objects. After the assignment, both bishop1 and bishop2 contain the same address, and therefore refer to the same object.

KEY CONCEPT
Several references can refer to the same object. These references are aliases of each other.

The bishop1 and bishop2 references are now *aliases* of each other, because they are two names that refer to the same object. All references to the object that was originally referenced by bishop2 are now gone; that object cannot be used again in the program.

One important implication of aliases is that when we use one reference to change the state of the object, it is also changed for the other, because there is really only one object. If you change the state of bishop1, for instance, you change the state of bishop2, because they both refer to the same object. Aliases can produce undesirable effects unless they are managed carefully.

Another important aspect of references is the way they affect how we determine if two objects are equal. The == operator that we use for primitive data can be used with object references, but it returns true only if the two references being compared

KEY CONCEPT

The == operator compares object references for equality, returning true if the references are aliases of each other.

are aliases of each other. It does not "look inside" the objects to see if they contain the same data.

That is, the following expression is true only if `bishop1` and `bishop2` currently refer to the same object:

```
bishop1 == bishop2
```

A method called `equals` is defined for all objects, but unless we replace it with a specific definition when we write a class, it has the same semantics as the == operator. That is, the `equals` method returns a `boolean` value that, by default, will be true if the two objects being compared are aliases of each other. The `equals` method is invoked through one object, and takes the other one as a parameter. Therefore, the expression

```
bishop1.equals(bishop2)
```

KEY CONCEPT

The `equals` method can be defined to determine equality between objects in any way we consider appropriate.

returns true if both references refer to the same object. However, we could define the `equals` method in the `ChessPiece` class to define equality for `ChessPiece` objects any way we would like. That is, we could define the `equals` method to return true under whatever conditions we think are appropriate to mean that one `ChessPiece` is equal to another.

The `equals` method has been given an appropriate definition in the `String` class. When comparing two `String` objects, the `equals` method returns true only if both strings contain the same characters. A common mistake is to use the == operator to compare strings, which compares the references for equality, when most of the time we want to compare the characters in the strings for equality. The `equals` method is discussed in more detail later in this chapter.

Garbage Collection

All interaction with an object occurs through a reference variable, so we can use an object only if we have a reference to it. When all references to an object are lost (perhaps by reassignment), that object can no longer participate in the program. The program can no longer invoke its methods or use its variables. At this point the object is called *garbage* because it serves no useful purpose.

KEY CONCEPT

If an object has no references to it, a program cannot use it. Java performs automatic garbage collection by periodically reclaiming the memory space occupied by these objects.

Java performs *automatic garbage collection*. When the last reference to an object is lost, the object becomes a candidate for garbage collection. Occasionally, the Java run time executes a method that "collects" all of the objects marked for garbage collection and returns their allocated memory to the system for future use. The programmer does not have to worry about explicitly returning memory that has become garbage.

If there is an activity that a programmer wants to accomplish in conjunction with the object being destroyed, the programmer can define a method called `finalize` in the object's class. The `finalize` method takes no parameters and has a `void` return type. It will be executed by the Java run time after the object is marked for garbage collection and before it is actually destroyed. The `finalize` method is not often used because the garbage collector performs most normal cleanup operations. However, it is useful for performing activities that the garbage collector does not address, such as closing files.

Passing Objects as Parameters

Another important issue related to object references comes up when we want to pass an object to a method. Java passes all parameters to a method *by value*. That is, the current value of the actual parameter (in the invocation) is copied into the formal parameter in the method header. Essentially, parameter passing is like an assignment statement, assigning to the formal parameter a copy of the value stored in the actual parameter.

This issue must be considered when making changes to a formal parameter inside a method. The formal parameter is a separate copy of the value that is passed in, so any changes made to it have no effect on the actual parameter. After control returns to the calling method, the actual parameter will have the same value as it did before the method was called.

However, when we pass an object to a method, we are actually passing a reference to that object. The value that gets copied is the address of the object. Therefore, the formal parameter and the actual parameter become aliases of each other. If we change the state of the object through the formal parameter reference inside the method, we are changing the object referenced by the actual parameter, because they refer to the same object. On the other hand, if we change the formal parameter reference itself (to make it point to a new object, for instance), we have not changed the fact that the actual parameter still refers to the original object.

KEY CONCEPT

When an object is passed to a method, the actual and formal parameters become aliases of each other.

B.10 The `static` Modifier

We have seen how visibility modifiers allow us to specify the encapsulation characteristics of variables and methods in a class. Java has several other modifiers that determine other characteristics. For example, the `static` modifier associates a variable or method with its class rather than with an object of the class.

Static Variables

So far, we have seen two categories of variables: local variables, which are declared inside a method, and instance variables, which are declared in a class but not inside a method. The term *instance variable* is used because an instance variable is accessed through a particular instance (an object) of a class. In general, each object has distinct memory space for each variable, so that each object can have a distinct value for that variable.

Another kind of variable, called a *static variable* or *class variable*, is shared among all instances of a class. There is only one copy of a static variable for all objects of a class. Therefore, changing the value of a static variable in one object changes it for all of the others. The reserved word `static` is used as a modifier to declare a static variable:

```
private static int count = 0;
```

KEY CONCEPT
A static variable is shared among all instances of a class.

Memory space for a static variable is established when the class that contains it is referenced for the first time in a program. A local variable declared within a method cannot be static.

Constants, which are declared using the `final` modifier, are also often declared using the `static` modifier as well. Because the value of constants cannot be changed, there might as well be only one copy of the value across all objects of the class.

Static Methods

A *static method* (also called a *class method*) can be invoked through the class name (all the methods of the `Math` class are static methods, for example). You don't have to instantiate an object of the class to invoke a static method. For example, the `sqrt` method is called through the `Math` class as follows:

```
System.out.println ("Square root of 27: " + Math.sqrt(27));
```

KEY CONCEPT
A method is made static by using the `static` modifier in the method declaration.

A method is made static by using the `static` modifier in the method declaration. As we have seen, the `main` method of a Java program must be declared with the `static` modifier; this is so that `main` can be executed by the interpreter without instantiating an object from the class that contains `main`.

Because static methods do not operate in the context of a particular object, they cannot reference instance variables, which exist only in an instance of a class. The compiler will issue an error if a static method attempts to use a nonstatic variable. A static method can, however, reference static variables, because static variables exist independent of specific objects. Therefore, the `main` method can access only static or local variables.

The methods in the `Math` class perform basic computations based on values passed as parameters. There is no object state to maintain in these situations; therefore, there is no good reason to force us to create an object in order to request these services.

B.11 Wrapper Classes

In some object-oriented programming languages, everything is represented using classes and the objects that are instantiated from them. In Java there are primitive types (such as `int`, `double`, `char`, and `boolean`) in addition to classes and objects.

Having two categories of data to manage (primitive values and object references) can present a challenge in some circumstances. For example, we might create an object that serves as a collection to hold various types of other objects. But in a specific situation we want the collection to hold simple integer values. In these cases we need to "wrap" a primitive type into a class so that it can be treated as an object.

A *wrapper class* represents a particular primitive type. For instance, the `Integer` class represents a simple integer value. An object created from the `Integer` class stores a single `int` value. The constructors of the wrapper classes accept the primitive value to store. For example:

```
Integer ageObj = new Integer(45);
```

Once this declaration and instantiation are performed, the `ageObj` object effectively represents the integer 45 as an object. It can be used wherever an object is called for in a program instead of a primitive type.

For each primitive type in Java there exists a corresponding wrapper class in the Java class library. All wrapper classes are defined in the `java.lang` package. There is even a wrapper class that represents the type `void`. However, unlike the other wrapper classes, the `Void` class cannot be instantiated. It simply represents the concept of a void reference.

KEY CONCEPT

A wrapper class represents a primitive value so that it can be treated as an object.

The wrapper classes also provide various methods related to the management of the associated primitive type. For example, the `Integer` class contains methods that return the `int` value stored in the object, and that convert the stored value to other primitive types.

Wrapper classes also contain static methods that can be invoked independent of any instantiated object. For example, the `Integer` class contains a static method called `parseInt` to convert an integer that is stored in a `String` to its corresponding `int` value. If the `String` object `str` holds the string "`987`", then the following line of code converts and stores the integer value `987` into the `int` variable `num`:

```
num = Integer.parseInt(str);
```

The Java wrapper classes often contain static constants that are helpful as well. For example, the `Integer` class contains two constants, `MIN_VALUE` and `MAX_VALUE`, which hold the smallest and largest `int` values, respectively. The other wrapper classes contain similar constants for their types.

B.12 Interfaces

We have used the term "interface" to mean the public methods through which we can interact with an object. That definition is consistent with our use of it in this section, but now we are going to formalize this concept using a particular language construct in Java.

KEY CONCEPT
An interface is a collection of abstract methods. It cannot be instantiated.

A Java *interface* is a collection of constants and abstract methods. An *abstract method* is a method that does not have an implementation. That is, there is no body of code defined for an abstract method. The header of the method, including its parameter list, is simply followed by a semicolon. An interface cannot be instantiated.

The following interface, called `Complexity`, contains two abstract methods, `setComplexity` and `getComplexity`:

```
interface Complexity
{
   void setComplexity (int complexity);
   int getComplexity ();
}
```

An abstract method can be preceded by the reserved word `abstract`, though in interfaces it usually is not. Methods in interfaces have public visibility by default.

KEY CONCEPT
A class implements an interface, which formally defines a set of methods used to interact with objects of that class.

A class *implements* an interface by providing method implementations for each of the abstract methods defined in the interface. A class that implements an interface uses the reserved word `implements` followed by the interface name in the class header. If a class asserts that it implements a particular interface, it must provide a definition for all methods in the interface. The compiler will produce errors if any of the methods in the interface is not given a definition in the class.

For example, a class called `Question` could be defined to represent a question that a teacher may ask on a test. If the `Question` class implements the `Complexity` interface, it must explicitly say so in the header and must define both methods from the `Complexity` interface:

```
class Questions implements Complexity
{
   int difficulty;
   // whatever else
   void setComplexity (int complexity)
   {
      difficulty = complexity;
   }
```

```
      int getComplexity ()
      {
         return difficulty;
      }
   }
```

Multiple classes can implement the same interface, providing alternative definitions for the methods. For example, we could implement a class called `Task` that also implements the `Complexity` interface. In it we could choose to manage the complexity of a task in a different way (though it would still have to implement all the methods of the interface).

A class can implement more than one interface. In these cases, the class must provide an implementation for all methods in all interfaces listed. To show that a class implements multiple interfaces, they are listed in the `implements` clause, separated by commas. For example:

```
class ManyThings implements interface1, interface2, interface3
{
   // all methods of all interfaces
}
```

In addition to, or instead of, abstract methods, an interface can also contain constants, defined using the `final` modifier. When a class implements an interface, it gains access to all of the constants defined in it. This mechanism allows multiple classes to share a set of constants that are defined in a single location.

The Comparable Interface

The Java standard class library contains interfaces as well as classes. The `Comparable` interface, for example, is defined in the `java.lang` package. It contains only one method, `compareTo`, which takes an object as a parameter and returns an integer.

The intention of this interface is to provide a common mechanism for comparing one object to another. One object calls the method and passes another as a parameter:

```
if (obj1.compareTo(obj2) < 0)
   System.out.println ("obj1 is less than obj2");
```

As specified by the documentation for the interface, the integer that is returned from the `compareTo` method should be negative if `obj1` is less than `obj2`, 0 if they are equal, and positive if `obj1` is greater than `obj2`. It is up to the designer of each class to decide what it means for one object of that class to be less than, equal to, or greater than another.

The `String` class contains a `compareTo` method that operates in this manner. Now we can clarify that the `String` class has this method because it implements the `Comparable` interface. The `String` class implementation of this method bases the comparison on the lexicographic ordering defined by the Unicode character set.

The `Iterator` Interface

The `Iterator` interface is another interface defined as part of the Java standard class library. It is used by classes that represent a collection of objects, providing a means to move through the collection one object at a time.

The two primary methods in the `Iterator` interface are `hasNext`, which returns a `boolean` result, and `next`, which returns an object. Neither of these methods takes any parameters. The `hasNext` method returns true if there are items left to process, and `next` returns the next object. It is up to the designer of the class that implements the `Iterator` interface to decide the order in which objects will be delivered by the `next` method.

We should note that, according to the spirit of the interface, the `next` method does not remove the object from the underlying collection; it simply returns a reference to it. The `Iterator` interface also has a method called `remove`, which takes no parameters and has a `void` return type. A call to the `remove` method removes the object that was most recently returned by the `next` method from the underlying collection.

The `Iterator` interface is an improved version of an older interface called `Enumeration`, which is still part of the Java standard class library. The `Enumeration` interface does not have a remove method. Generally, the `Iterator` interface is the preferred choice between the two.

B.13 Inheritance

A class establishes the characteristics and behaviors of an object, but reserves no memory space for variables (unless those variables are declared as static). Classes are the plan, and objects are the embodiment of that plan.

Many houses can be created from the same blueprint. They are essentially the same house in different locations with different people living in them. But suppose you want a house that is similar to another, but with some different or additional features. You want to start with the same basic blueprint but modify it to suit your needs and desires. Many housing developments are created this way. The houses in the development have the same core layout, but they can have unique features. For instance, they might all be split-level homes with the same bedroom, kitchen, and living-room configuration, but some have a fireplace or full basement whereas others do not, and some have an attached garage instead of a carport.

It's likely that the housing developer commissioned a master architect to create a single blueprint to establish the basic design of all houses in the development, then a series of new blueprints that include variations designed to appeal to different buyers. The act of creating the series of blueprints was simplified because they all begin with the same underlying structure, while the variations give them unique characteristics that may be very important to the prospective owners.

Creating a new blueprint that is based on an existing blueprint is analogous to the object-oriented concept of *inheritance*, which allows a software designer to define a new class in terms of an existing one. It is a powerful software development technique and a defining characteristic of object-oriented programming.

Derived Classes

Inheritance is the process in which a new class is derived from an existing one. The new class automatically contains some or all of the variables and methods in the original class. Then, to tailor the class as needed, the programmer can add new variables and methods to the derived class, or modify the inherited ones.

KEY CONCEPT

Inheritance is the process of deriving a new class from an existing one.

In general, creating new classes via inheritance is faster, easier, and cheaper than writing them from scratch. At the heart of inheritance is the idea of *software reuse*. By using existing software components to create new ones, we capitalize on all of the effort that went into the design, implementation, and testing of the existing software.

KEY CONCEPT

One purpose of inheritance is to reuse existing software.

Keep in mind that the word *class* comes from the idea of classifying groups of objects with similar characteristics. Classification schemes often use levels of classes that relate to one another. For example, all mammals share certain characteristics: they are warm-blooded, have hair, and bear live offspring. Now consider a subset of mammals, such as horses. All horses are mammals and have all the characteristics of mammals. But they also have unique features that make them different from other mammals.

If we map this idea into software terms, an existing class called `Mammal` would have certain variables and methods that describe the state and behavior of mammals. A `Horse` class could be derived from the existing `Mammal` class, automatically inheriting the variables and methods contained in `Mammal`. The `Horse` class can refer to the inherited variables and methods as if they had been declared locally in that class. New variables and methods can then be added to the derived class, to distinguish a horse from other mammals. Inheritance nicely models many situations found in the natural world.

KEY CONCEPT
Inherited variables and methods can be used in the derived class as if they had been declared locally.

KEY CONCEPT
Inheritance creates an is-a relationship between all parent and child classes.

The original class that is used to derive a new one is called the *parent class*, *superclass*, or *base class*. The derived class is called a *child class*, or *subclass*. Java uses the reserved word `extends` to indicate that a new class is being derived from an existing class.

The derivation process should establish a specific kind of relationship between two classes: an *is-a relationship*. This type of relationship means that the derived class should be a more specific version of the original. For example, a horse is a mammal. Not all mammals are horses, but all horses are mammals.

Let's look at an example. The following class can be used to define a book:

```
class Book
{
   protected int numPages;

   protected void pages()
   {
      System.out.println ("Number of pages: " + numPages);
   }
}
```

To derive a child class that is based on the `Book` class, we use the reserved word `extends` in the header of the child class. For example, a `Dictionary` class can be derived from `Book` as follows:

```
class Dictionary extends Book
{
   private int numDefs;

   public void info()
   {
      System.out.println ("Number of definitions: " + numDefs);
      System.out.println ("Definitions per page: "
                          + numDefs/numPages);
   }
}
```

By saying that the `Dictionary` class extends the `Book` class, the `Dictionary` class automatically inherits the `numPages` variable and the `pages` method. Note that the `info` method uses the `numPages` variable explicitly.

Inheritance is a one-way street. The `Book` class cannot use variables or methods that are declared explicitly in the `Dictionary` class. For instance, if we created an object from the `Book` class, it could not be used to invoke the `info` method. This restriction makes sense, because a child class is a more specific version of the parent.

A dictionary has pages, because all books have pages; but although a dictionary has definitions, not all books do.

Inheritance relationships are represented in UML class diagrams using an arrow with an open arrowhead pointing from the child class to the parent class.

The `protected` Modifier

Not all variables and methods are inherited in a derivation. The visibility modifiers used to declare the members of a class determine which ones are inherited and which are not. Specifically, the child class inherits variables and methods that are declared public, and it does not inherit those that are declared private.

However, if we declare a variable with public visibility so that a derived class can inherit it, we violate the principle of encapsulation. Therefore, Java provides a third visibility modifier: `protected`. When a variable or method is declared with protected visibility, a derived class will inherit it, retaining some of its encapsulation properties. The encapsulation with protected visibility is not as tight as it would be if the variable or method were declared private, but it is better than if it were declared public. Specifically, a variable or method declared with protected visibility may be accessed by any class in the same package.

KEY CONCEPT

Visibility modifiers determine which variables and methods are inherited. Protected visibility provides the best possible encapsulation that permits inheritance.

Each inherited variable or method retains the effect of its original visibility modifier. For example, if a method is public in the parent, it is public in the child.

Constructors are not inherited in a derived class, even though they have public visibility. This is an exception to the rule about public members being inherited. Constructors are special methods that are used to set up a particular type of object, so it wouldn't make sense for a class called `Dictionary` to have a constructor called `Book`.

The `super` Reference

The reserved word `super` can be used in a class to refer to its parent class. Using the `super` reference, we can access a parent's members, even if they aren't inherited. Like the `this` reference, what the word `super` refers to depends on the class in which it is used. However, unlike the `this` reference, which refers to a particular instance of a class, `super` is a general reference to the members of the parent class.

One use of the `super` reference is to invoke a parent's constructor. If the following invocation is performed at the beginning of a constructor, the parent's constructor is invoked, passing any appropriate parameters:

```
super (x, y, z);
```

A child's constructor is responsible for calling its parent's constructor. Generally, the first line of a constructor should use the `super` reference call to a constructor of

KEY CONCEPT
A parent's constructor can be invoked using the `super` reference.

the parent class. If no such call exists, Java will automatically make a call to `super()` at the beginning of the constructor. This rule ensures that a parent class initializes its variables before the child class constructor begins to execute. Using the `super` reference to invoke a parent's constructor can be done only in the child's constructor and, if included, must be the first line of the constructor.

The `super` reference can also be used to reference other variables and methods defined in the parent's class.

Overriding Methods

When a child class defines a method with the same name and signature as a method in the parent, we say that the child's version *overrides* the parent's version in favor of its own. The need for overriding occurs often in inheritance situations.

KEY CONCEPT
A child class can override (redefine) the parent's definition of an inherited method.

The object that is used to invoke a method determines which version of the method is actually executed. If it is an object of the parent type, the parent's version of the method is invoked. If it is an object of the child type, the child's version is invoked. This flexibility allows two objects that are related by inheritance to use the same naming conventions for methods that accomplish the same general task in different ways.

A method can be defined with the `final` modifier. A child class cannot override a final method. This technique is used to ensure that a derived class uses a particular definition for a method.

The concept of method overriding is important to several issues related to inheritance. These issues are explored in later sections of this chapter.

B.14 Class Hierarchies

A child class derived from one parent can be the parent of its own child class. Furthermore, multiple classes can be derived from a single parent. Therefore, inheritance relationships often develop into *class hierarchies*. The UML class diagram in Figure B.4 shows a class hierarchy that incorporates the inheritance relationship between classes `Mammal` and `Horse`.

KEY CONCEPT
The child of one class can be the parent of one or more other classes, creating a class hierarchy.

There is no limit to the number of children a class can have, or to the number of levels to which a class hierarchy can extend. Two children of the same parent are called *siblings*. Although siblings share the characteristics passed on by their common parent, they are not related by inheritance, because one is not used to derive the other.

In class hierarchies, common features should be kept as high in the hierarchy as reasonably possible. That way, the only characteristics

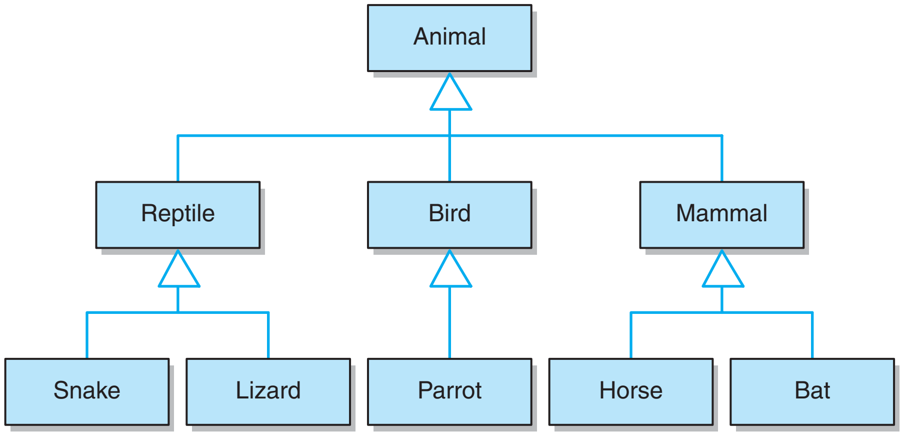


FIGURE B.4 A UML class diagram showing a class hierarchy

explicitly established in a child class are those that make the class distinct from its parent and from its siblings. This approach maximizes the ability to reuse classes. It also facilitates maintenance activities, because when changes are made to the parent, they are automatically reflected in the descendants. Always remember to maintain the is-a relationship when building class hierarchies.

KEY CONCEPT
Common features should be located as high in a class hierarchy as is reasonable, minimizing maintenance efforts.

The inheritance mechanism is transitive. That is, a parent passes along a trait to a child class, that child class passes it along to its children, and so on. An inherited feature might have originated in the immediate parent, or possibly from several levels higher in a more distant ancestor class.

There is no single best hierarchy organization for all situations. The decisions made when designing a class hierarchy restrict and guide more detailed design decisions and implementation options, and they must be made carefully.

The `Object` Class

In Java, all classes are derived ultimately from the `Object` class. If a class definition doesn't use the `extends` clause to derive itself explicitly from another class, then that class is automatically derived from the `Object` class by default. Therefore, the following two class definitions are equivalent:

```
class Thing
{
   // whatever
}
```

and

```
class Thing extends Object
{
   // whatever
}
```

Because all classes are derived from `Object`, any public method of `Object` can be invoked through any object created in any Java program. The `Object` class is defined in the `java.lang` package of the standard class library.

KEY CONCEPT
All Java classes are derived, directly or indirectly, from the `Object` class.

The `toString` method, for instance, is defined in the `Object` class, so the `toString` method can be called on any object. When a `println` method is called with an object parameter, `toString` is called to determine what to print.

The definition for `toString` that is provided by the `Object` class returns a string containing the object's class name followed by a numeric value that is unique for that object. Usually, we override the `Object` version of `toString` to fit our own needs. The `String` class has overridden the `toString` method so that it returns its stored string value.

KEY CONCEPT
The `toString` and `equals` methods are defined in the `Object` class and therefore are inherited by every class in every Java program.

The `equals` method of the `Object` class is also useful. Its purpose is to determine if two objects are equal. The definition of the `equals` method provided by the `Object` class returns true if the two object references actually refer to the same object (that is, if they are aliases). Classes often override the inherited definition of the `equals` method in favor of a more appropriate definition. For instance, the `String` class overrides `equals` so that it returns true only if both strings contain the same characters in the same order.

Abstract Classes

An *abstract class* represents a generic concept in a class hierarchy. An abstract class cannot be instantiated and usually contains one or more abstract methods, which have no definition. In this sense, an abstract class is similar to an interface. Unlike interfaces, however, an abstract class can contain methods that are not abstract, and can contain data declarations other than constants.

A class is declared as abstract by including the `abstract` modifier in the class header. Any class that contains one or more abstract methods must be declared as abstract. In abstract classes (unlike interfaces), the `abstract` modifier must be applied to each abstract method. A class declared as abstract does not have to contain abstract methods.

Abstract classes serve as placeholders in a class hierarchy. As the name implies, an abstract class represents an abstract entity that is usually insufficiently defined to

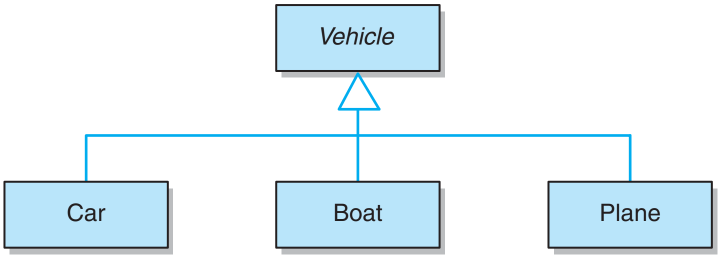


FIGURE B.5 A `Vehicle` class hierarchy

be useful by itself. Instead, an abstract class may contain a partial description that is inherited by all of its descendants in the class hierarchy. Its children, which are more specific, fill in the gaps.

KEY CONCEPT
An abstract class cannot be instantiated. It represents a concept on which other classes can build their definitions.

Consider the class hierarchy shown in Figure B.5. The `Vehicle` class at the top of the hierarchy may be too generic for a particular application. Therefore, we may choose to implement it as an abstract class. Concepts that apply to all vehicles can be represented in the `Vehicle` class and are inherited by its descendants. That way, each of its descendants doesn't have to define the same concept redundantly, and perhaps inconsistently.

For example, we may say that all vehicles have a particular speed. Therefore, we declare a `speed` variable in the `Vehicle` class, and all specific vehicles below it in the hierarchy automatically have that variable via inheritance. Any change we make to the representation of the speed of a vehicle is automatically reflected in all descendant classes. Similarly, we may declare an abstract method called `fuelConsumption`, whose purpose is to calculate how quickly fuel is being consumed by a particular vehicle. The details of the `fuelConsumption` method must be defined by each type of vehicle, but the `Vehicle` class establishes that all vehicles consume fuel and provides a consistent way to compute that value.

Some concepts don't apply to all vehicles, so we wouldn't represent those concepts at the `Vehicle` level. For instance, we wouldn't include a variable called `numberOfWheels` in the `Vehicle` class, because not all vehicles have wheels. The child classes for which wheels are appropriate can add that concept at the appropriate level in the hierarchy.

There are no restrictions as to where in a class hierarchy an abstract class can be defined. Usually they are located at the upper levels of a class hierarchy. However, it is possible to derive an abstract class from a nonabstract parent.

Usually, a child of an abstract class will provide a specific definition for an abstract method inherited from its parent. Note that this is just a specific case of overriding a method, giving a different definition than the one the parent provides. If a

KEY CONCEPT
A class derived from an abstract parent must override all of its parent's abstract methods, or the derived class will also be considered abstract.

child of an abstract class does not give a definition for every abstract method that it inherits from its parent, then the child class is also considered to be abstract.

Note that it would be a contradiction for an abstract method to be modified as `final` or `static`. Because a final method cannot be overridden in subclasses, an abstract final method would have no way of being given a definition in subclasses. A static method can be invoked using the class name without declaring an object of the class. Because abstract methods have no implementation, an abstract static method would make no sense.

Choosing which classes and methods to make abstract is an important part of the design process. Such choices should be made only after careful consideration. By using abstract classes wisely, we can create flexible, extensible software designs.

Interface Hierarchies

The concept of inheritance can be applied to interfaces as well as classes. That is, one interface can be derived from another interface. These relationships can form an *interface hierarchy*, which is similar to a class hierarchy. Inheritance relationships between interfaces are shown in UML using the same connection (an arrow with an open arrowhead) as they are with classes.

KEY CONCEPT
Inheritance can be applied to interfaces, so that one interface can be derived from another interface.

When a parent interface is used to derive a child interface, the child inherits all abstract methods and constants of the parent. Any class that implements the child interface must implement all of the methods. There are no restrictions on the inheritance between interfaces, as there are with protected and private members of a class, because all members of an interface are public.

Class hierarchies and interface hierarchies do not overlap. That is, an interface cannot be used to derive a class, and a class cannot be used to derive an interface. A class and an interface interact only when a class is designed to implement a particular interface.

B.15 Polymorphism

Usually, the type of a reference variable matches the class of the object it refers to exactly. That is, if we declare a reference as follows:

```
ChessPiece bishop;
```

the `bishop` reference is used to refer to an object created by instantiating the `ChessPiece` class. However, the relationship between a reference variable and the object it refers to is more flexible than that.

The term *polymorphism* can be defined as “having many forms.” A *polymorphic reference* is a reference variable that can refer to different types of objects at different points in time. The specific method invoked through a polymorphic reference can change from one invocation to the next.

KEY CONCEPT
A polymorphic reference can refer to different types of objects over time.

Consider the following line of code:

```
obj.doIt();
```

If the reference `obj` is polymorphic, it can refer to different types of objects at different times. If that line of code is in a loop or in a method that is called more than once, that line of code might call a different version of the `doIt` method each time it is invoked.

At some point, the commitment is made to execute certain code to carry out a method invocation. This commitment is referred to as *binding* a method invocation to a method definition. In most situations, the binding of a method invocation to a method definition can occur at compile time. For polymorphic references, however, the decision cannot be made until run time. The method definition that is used is based on the object that is being referred to by the reference variable at that moment. This deferred commitment is called *late binding* or *dynamic binding*. It is less efficient than binding at compile time because the decision has to be made during the execution of the program. This overhead is generally acceptable in light of the flexibility that a polymorphic reference provides.

There are two ways to create a polymorphic reference in Java: using inheritance and using interfaces. The following sections describe these approaches.

References and Class Hierarchies

In Java, a reference that is declared to refer to an object of a particular class also can be used to refer to an object of any class related to it by inheritance. For example, if the class `Mammal` is used to derive the class `Horse`, then a `Mammal` reference can be used to refer to an object of class `Horse`. This ability is shown in the code segment below:

```
Mammal pet;
Horse secretariat = new Horse();
pet = secretariat; // a valid assignment
```

The reverse operation, assigning the `Mammal` object to a `Horse` reference, is also valid, but requires an explicit cast. Assigning a reference in this direction is generally less useful and more likely to cause problems, because although a horse has all the functionality of a mammal (because a horse *is-a* mammal), the reverse is not necessarily true.

KEY CONCEPT
A reference variable can refer to any object created from any class related to it by inheritance.

This relationship works throughout a class hierarchy. If the `Mammal` class were derived from a class called `Animal`, then the following assignment would also be valid:

```
Animal creature = new Horse();
```

Carrying this to the extreme, an `Object` reference can be used to refer to any object, because ultimately all classes are descendants of the `Object` class. An `ArrayList`, for example, uses polymorphism in that it is designed to hold `Object` references. That's why an `ArrayList` can be used to store any kind of object. In fact, a particular `ArrayList` can be used to hold several different types of objects at one time, because, in essence, they are all `Object` objects.

Polymorphism via Inheritance

The reference variable `creature`, as defined in the previous section, can be polymorphic, because at any point in time it could refer to an `Animal` object, a `Mammal` object, or a `Horse` object. Suppose that all three of these classes have a method called `move` and that it is implemented in a different way in each class (because the child class overrode the definition it inherited). The following invocation calls the `move` method, but the particular version of the method it calls is determined at run time:

```
creature.move();
```

At the point when this line is executed, if `creature` currently refers to an `Animal` object, the `move` method of the `Animal` class is invoked. Likewise, if creature currently refers to a `Mammal` or `Horse` object, the `Mammal` or `Horse` version of `move` is invoked, respectively.

KEY CONCEPT

A polymorphic reference uses the type of the object, not the type of the reference, to determine which version of a method to invoke.

Of course, because `Animal` and `Mammal` represent general concepts, they may be defined as abstract classes. This situation does not eliminate the ability to have polymorphic references. Suppose the `move` method in the `Mammal` class is abstract, and is given unique definitions in the `Horse`, `Dog`, and `Whale` classes (all derived from `Mammal`). A `Mammal` reference variable can be used to refer to any objects created from any of the `Horse`, `Dog`, and `Whale` classes, and can be used to execute the `move` method on any of them.

Let's consider another situation. The class hierarchy shown in Figure B.6 contains classes that represent various types of employees that might work at a particular company.

Polymorphism could be used in this situation to pay various employees in different ways. One list of employees (of whatever type) could be paid using a single loop that invokes each employee's pay method. But the pay method that is invoked each time will depend on the specific type of employee that is executing the pay method during that iteration of the loop.

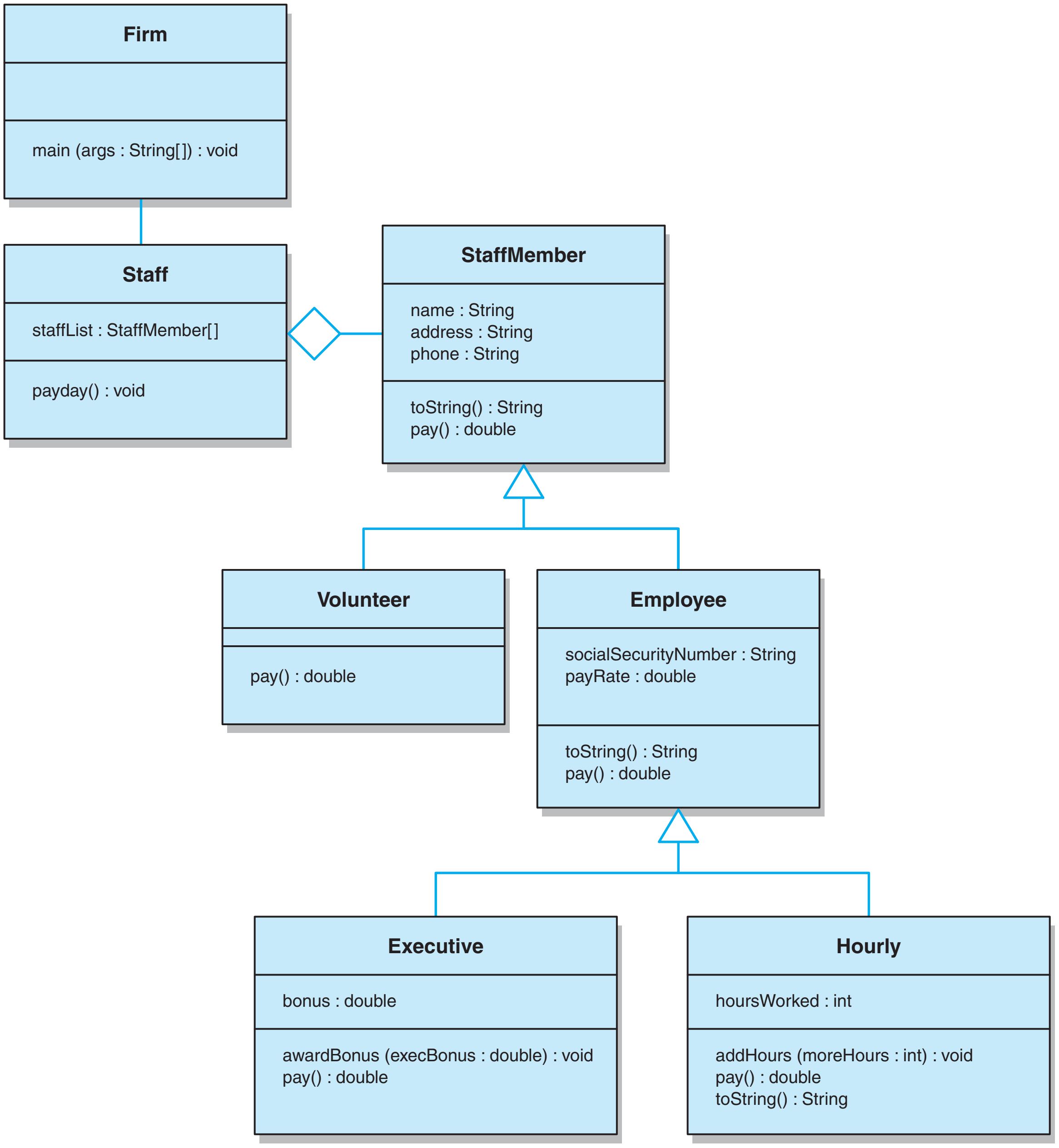


FIGURE B.6 A class hierarchy of employees

This is a classic example of polymorphism—allowing different types of objects to handle a similar operation in different ways.

Polymorphism via Interfaces

As we have seen, a class name is used to declare the type of an object reference variable. Similarly, an interface name can be used as the type of a reference variable as well. An interface reference variable can be used to refer to any object of any class that implements that interface.

Suppose we declare an interface called `Speaker` as follows:

```
public interface Speaker
{
   public void speak();
   public void announce (String str);
}
```

KEY CONCEPT

An interface name can be used to declare an object reference variable. An interface reference can refer to any object of any class that implements the interface.

The interface name, `Speaker`, can now be used to declare an object reference variable:

```
Speaker current;
```

The reference variable `current` can be used to refer to any object of any class that implements the `Speaker` interface. For example, if we define a class called `Philosopher` such that it implements the `Speaker` interface, we could then assign a `Philosopher` object to a `Speaker` reference:

```
current = new Philosopher();
```

This assignment is valid because a `Philosopher` is, in fact, a `Speaker`.

KEY CONCEPT

Interfaces allow us to make polymorphic references, in which the method that is invoked is based on the particular object being referenced at the time.

The flexibility of an interface reference allows us to create polymorphic references. As we saw earlier in this chapter, by using inheritance, we can create a polymorphic reference that can refer to any one of a set of objects related by inheritance. Using interfaces, we can create similar polymorphic references, except that the objects being referenced are related by implementing the same interface instead of being related by inheritance.

For example, if we create a class called `Dog` that also implements the `Speaker` interface, it too could be assigned to a `Speaker` reference variable. The same reference, in fact, could at one point refer to a `Philosopher` object and then later refer to a `Dog` object. The following lines of code illustrate this:

```
Speaker guest;
guest = new Philosopher();
guest.speak();
guest = new Dog();
guest.speak();
```

In this code, the first time the speak method is called, it invokes the speak method defined in the Philosopher class. The second time it is called, it invokes the speak method of the Dog class. As with polymorphic references via inheritance, it is not the type of the reference that determines which method gets invoked, but rather it is the type of the object that the reference points to at the moment of invocation.

Note that when we are using an interface reference variable, we can invoke only the methods defined in the interface, even if the object it refers to has other methods to which it can respond. For example, suppose the Philosopher class also defined a public method called pontificate. The second line of the following code would generate a compiler error, even though the object can in fact respond to the pontificate method:

```
Speaker special = new Philosopher();
special.pontificate(); // generates a compiler error
```

The problem is that the compiler can determine only that the object is a Speaker, and therefore can guarantee only that the object can respond to the speak and announce methods. Because the reference variable special could refer to a Dog object (which cannot pontificate), it does not allow the reference. If we know in a particular situation that such an invocation is valid, we can cast the object into the appropriate reference so that the compiler will accept it:

```
((Philosopher) special).pontificate();
```

Similar to polymorphic references based on inheritance, an interface name can be used as the type of a method parameter. In such situations, any object of any class that implements the interface can be passed into the method. For example, the following method takes a Speaker object as a parameter. Therefore, both a Dog object and a Philosopher object can be passed into it in separate invocations.

```
public void sayIt (Speaker current)
{
   current.speak();
}
```

B.16 Generic Types

Java enables us to define a class based on a *generic type*. That is, we can define a class so that it stores, operates on, and manages objects whose type is not specified until the class is instantiated. Generics are an integral part of our discussions of collections and their underlying implementations throughout the rest of this book.

KEY CONCEPT
Errors and exceptions represent unusual or invalid processing.

Let's assume we need to define a class called `Box` that stores and manages other objects. Using polymorphism, we could simply define `Box` so that internally it stores references to the `Object` class. Then, any type of object could be stored inside a box. In fact, multiple types of unrelated objects could be stored in `Box`. We lose a lot of control with that level of flexibility in our code.

A better approach is to define the `Box` class to store a generic type `T`. (We can use any identifier we want for the generic type, though using `T` has become a convention.) The header of the class contains a reference to the type in angle brackets. For example:

```
class Box<T>
{
   // declarations and code that manage objects of type T
}
```

Then, when a `Box` is needed, it is instantiated with a specific class used in place of `T`. For example, if we wanted a `Box` of `Widget` objects, we could use the following declaration:

```
Box<Widget> box1 = new Box<Widget>;
```

The type of the `box1` variable is `Box<Widget>`. In essence, for the `box1` object, the `Box` class replaces `T` with `Widget`. Now suppose we wanted a `Box` in which to store `Gadget` objects; we could make the following declaration:

```
Box<Gadget> box2 = new Box<Gadget>;
```

For `box2`, the `Box` class essentially replaces `T` with `Gadget`. So, although the `box1` and `box2` objects are both boxes, they have different types because the generic type is taken into account. This is a safer implementation, because at this point we cannot use `box1` to store gadgets (or anything else for that matter), nor could we use `box2` to store widgets.

A generic type such as `T` cannot be instantiated. It is merely a placeholder to allow us to define the class that will manage a specific type of object that is established when the class is instantiated.

B.17 Exceptions

Problems that arise in a Java program may generate exceptions or errors. An *exception* is an object that defines an unusual or erroneous situation. An exception is thrown by a program or the runtime environment, and it can be caught and handled appropriately if desired. An *error* is similar to an exception, except that an error generally represents an unrecoverable situation, and it should not be caught. Java has a predefined set of exceptions and errors that may occur during the execution of a program.

KEY CONCEPT

Errors and exceptions represent unusual or invalid processing.

A program can be designed to process an exception in one of three ways:

- Not handle the exception at all.
- Handle the exception where it occurs.
- Handle the exception at another point in the program.

We explore each of these approaches in the following sections.

Exception Messages

If an exception is not handled at all by the program, the program will terminate (abnormally) and produce a message that describes what exception occurred and where in the program it was produced. The information associated with an exception is often helpful in tracking down the cause of a problem.

Let's look at the output of an exception. An `ArithmeticException` is thrown when an invalid arithmetic operation is attempted, such as dividing by zero. When that exception is thrown, if there is no code in the program to handle the exception explicitly, the program terminates and prints a message similar to the following:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Zero.main (Zero.java:17)
```

The first line of the exception output indicates which exception was thrown and provides some information about why it was thrown. The remaining line or lines are the *call stack trace*, which indicates where the exception occurred. In this case, there is only one line in the call stack trace, but in other cases there may be several, depending on where the exception originated in the program.

KEY CONCEPT

The messages printed by a thrown exception indicate the nature of the problem and provide a method call stack trace.

The first line of the trace indicates the method, file, and line number where the exception occurred. The other lines in the trace, if present, indicate the methods that were called to get to the method that produced the exception. In this program, there is only one method, and it produced the exception; therefore, there is only one line in the trace.

The call stack trace information is also available by calling methods of the exception object that is being thrown. The method `getMessage` returns a string explaining the reason the exception was thrown. The method `printStackTrace` prints the call stack trace.

The `try` Statement

Let's now examine how we catch and handle an exception when it is thrown. A *try statement* consists of a `try` block followed by one or more `catch` clauses. The `try` block is a group of statements that may throw an exception. A `catch` clause defines how a particular kind of exception is handled. A `try` block can have several `catch` clauses associated with it, each dealing with a particular kind of exception. A `catch` clause is sometimes called an *exception handler*.

Here is the general format of a `try` statement:

```
try
{
   // statements in the try block
}
catch (IOException exception)
{
   // statements that handle the I/O problem
}
catch (NumberFormatException exception)
{
   // statements that handle the number format problem
}
```

When a `try` statement is executed, the statements in the `try` block are executed. If no exception is thrown during the execution of the `try` block, processing continues with the statement following the `try` statement (after all of the `catch` clauses). This situation is the normal execution flow and should occur most of the time.

KEY CONCEPT
Each `catch` clause on a `try` statement handles a particular kind of exception that may be thrown within the `try` block.

If an exception is thrown at any point during the execution of the `try` block, control is immediately transferred to the appropriate exception handler if it is present. That is, control transfers to the first `catch` clause whose specified exception corresponds to the class of the exception that was thrown. After the statements in the `catch` clause are executed, control transfers to the statement after the entire `try` statement.

Exception Propagation

If an exception is not caught and handled where it occurs, control is immediately returned to the method that invoked the method that produced the exception. We can design our software so that the exception is caught and handled at this outer level. If it isn't caught there, control returns to the method that called it. This process is called *propagating the exception*.

Exception propagation continues until the exception is caught and handled, or until it is propagated out of the `main` method, which terminates the program and produces an exception message. To catch an exception at an outer level, the method that produces the exception must be invoked inside a `try` block that has an appropriate `catch` clause to handle it.

KEY CONCEPT
If an exception is not caught and handled where it occurs, it is propagated to the calling method.

A programmer must pick the most appropriate level at which to catch and handle an exception. There is no single best answer. It depends on the situation and the design of the system. Sometimes the right approach will be not to catch an exception at all and let the program terminate.

KEY CONCEPT
A programmer must carefully consider how exceptions should be handled, if at all, and at what level.

The Exception Class Hierarchy

The classes that define various exceptions are related by inheritance, creating a class hierarchy that is shown in part in Figure B.7.

The `Throwable` class is the parent of both the `Error` class and the `Exception` class. Many types of exceptions are derived from the `Exception` class, and these classes also have many children. Though these high-level classes are defined in the `java.lang` package, many child classes that define specific exceptions are part of several other packages. Inheritance relationships can span package boundaries.

We can define our own exceptions by deriving a new class from `Exception` or one of its descendants. The class we choose as the parent depends on what situation or condition the new exception represents.

KEY CONCEPT
A new exception is defined by deriving a new class from the `Exception` class or one of its descendants.

After creating the class that defines the exception, an object of that type can be created as needed. The `throw` statement is used to throw the exception. For example:

```
throw new MyException();
```

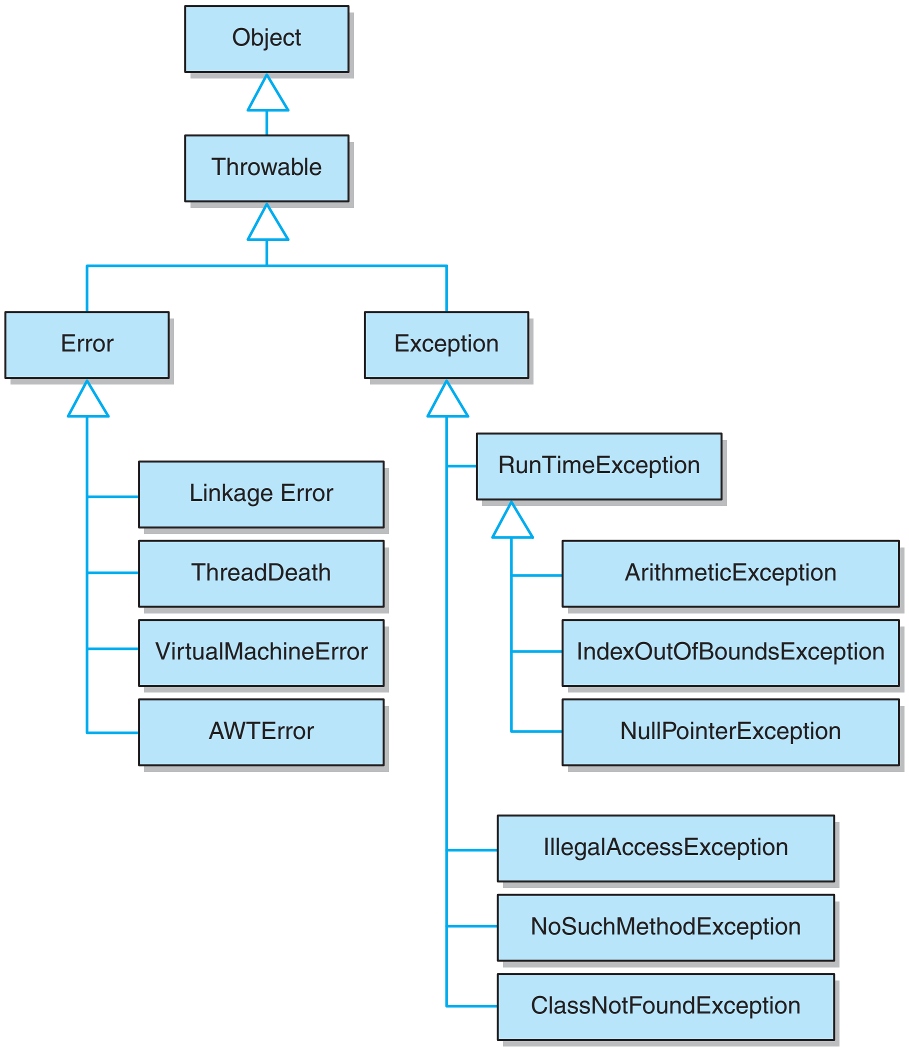


FIGURE B.7 Part of the `Error` and `Exception` class hierarchy

Summary of Key Concepts

- An abstraction hides details. A good abstraction hides the right details at the right time so that we can manage complexity.
- The `new` operator returns a reference to a newly created object.
- The Java standard class library is a useful set of classes that anyone can use when writing Java programs.
- A package is a Java language element used to group related classes under a common name.
- Each object has a state and a set of behaviors. The values of an object's variables define its state. The methods to which an object responds define its behaviors.
- A class is a blueprint for an object; it reserves no memory space for data. Each object has its own data space, and thus its own state.
- The scope of a variable, which determines where it can be referenced, depends on where it is declared.
- Objects should be encapsulated. The rest of a program should interact with an object only through a well-defined interface.
- Instance variables should be declared with private visibility to promote encapsulation.
- A variable declared in a method is local to that method and cannot be used outside of it.
- A constructor cannot have any return type, even `void`.
- The versions of an overloaded method are distinguished by their signatures. The number, type, or order of their parameters must be distinct.
- An object reference variable stores the address of an object.
- The reserved word `null` represents a reference that does not point to a valid object.
- The `this` reference always refers to the currently executing object.
- Several references can refer to the same object. These references are aliases of each other.
- The == operator compares object references for equality, returning true if the references are aliases of each other.
- The `equals` method can be defined to determine equality between objects in any way we consider appropriate.

- If an object has no references to it, a program cannot use it. Java performs automatic garbage collection by periodically reclaiming the memory space occupied by these objects.
- When an object is passed to a method, the actual and formal parameters become aliases of each other.
- A static variable is shared among all instances of a class.
- A method is made static by using the `static` modifier in the method declaration.
- A wrapper class represents a primitive value so that it can be treated as an object.
- An interface is a collection of abstract methods. It cannot be instantiated.
- A class implements an interface, which formally defines a set of methods used to interact with objects of that class.
- Inheritance is the process of deriving a new class from an existing one.
- One purpose of inheritance is to reuse existing software.
- Inherited variables and methods can be used in the derived class as if they had been declared locally.
- Inheritance creates an is-a relationship between all parent and child classes.
- Visibility modifiers determine which variables and methods are inherited. Protected visibility provides the best possible encapsulation that permits inheritance.
- A parent's constructor can be invoked using the `super` reference.
- A child class can override (redefine) the parent's definition of an inherited method.
- The child of one class can be the parent of one or more other classes, creating a class hierarchy.
- Common features should be located as high in a class hierarchy as is reasonable, minimizing maintenance efforts.
- All Java classes are derived, directly or indirectly, from the `Object` class.
- The `toString` and `equals` methods are defined in the `Object` class and therefore are inherited by every class in every Java program.
- An abstract class cannot be instantiated. It represents a concept on which other classes can build their definitions.
- A class derived from an abstract parent must override all of its parent's abstract methods, or the derived class will also be considered abstract.

- Inheritance can be applied to interfaces, so that one interface can be derived from another interface.
- A polymorphic reference can refer to different types of objects over time.
- A reference variable can refer to any object created from any class related to it by inheritance.
- A polymorphic reference uses the type of the object, not the type of the reference, to determine which version of a method to invoke.
- An interface name can be used to declare an object reference variable. An interface reference can refer to any object of any class that implements the interface.
- Interfaces allow us to make polymorphic references, in which the method that is invoked is based on the particular object being referenced at the time.
- Errors and exceptions represent unusual or invalid processing.
- The messages printed by a thrown exception indicate the nature of the problem and provide a method call stack trace.
- Each `catch` clause on a `try` statement handles a particular kind of exception that may be thrown within the `try` block.
- If an exception is not caught and handled where it occurs, it is propagated to the calling method.
- A programmer must carefully consider how exceptions should be handled, if at all, and at what level.
- A new exception is defined by deriving a new class from the `Exception` class or one of its descendants.

Self-Review Questions

SR B.1 What is the difference between an object and a class?

SR B.2 Objects should be self-governing. Explain.

SR B.3 Describe each of the following:

a. public method

b. private method

c. public variable

d. private variable

SR B.4 What are constructors used for? How are they defined?

SR B.5 How are overloaded methods distinguished from each other?

SR B.6 What is an aggregate object?

SR B.7 What is the difference between a static variable and an instance variable?

SR B.8 What is the difference between a class and an interface?

SR B.9 Describe the relationship between a parent class and a child class.

SR B.10 What relationship should every class derivation represent?

SR B.11 What is the significance of the `Object` class?

SR B.12 What is polymorphism?

SR B.13 How is overriding related to polymorphism?

SR B.14 How can polymorphism be accomplished using interfaces?

Exercises

EX B.1 Identify the following as a class, object, or method:

```
> superman
> breakChain
> SuperHero
> saveLife
```

EX B.2 Identify the following as a class, object, or method:

```
> Beverage
> pepsi
> drink
> refill
> coke
```

EX B.3 Explain why a static method cannot refer to an instance variable.

EX B.4 Can a class implement two interfaces that each contains the same method signature? Explain.

EX B.5 Describe the relationship between a parent class and a child class.

EX B.6 Draw and annotate a class hierarchy that represents various types of faculty at a university. Show what characteristics would be represented in the various classes of the hierarchy. Explain how polymorphism could play a role in the process of assigning courses to each faculty member.

Programming Projects

PP B.1 Design and implement a class called `Sphere` that contains instance data that represents the sphere's diameter. Define the `Sphere` constructor to accept and initialize the diameter, and include getter and setter methods for the diameter. Include methods that calculate and return the volume and surface area of the sphere (see Programming Project 3.2 for the formulas). Include a `toString` method that returns a one-line description of the sphere. Create a driver class called `MultiSphere`, whose `main` method instantiates and updates several `Sphere` objects.

PP B.2 Design and implement a class called `Dog` that contains instance data that represents the dog's name and age. Define the `Dog` constructor to accept and initialize instance data. Include getter and setter methods for the name and age. Include a method to compute and return the age of the dog in "person years" (seven times the dogs age). Include a `toString` method that returns a one-line description of the dog. Create a driver class called `Kennel`, whose `main` method instantiates and updates several `Dog` objects.

PP B.3 Design and implement a class called `Box` that contains instance data that represents the height, width, and depth of the box. Also include a `boolean` variable called `full` as instance data that represents if the box is full or not. Define the `Box` constructor to accept and initialize the height, width, and depth of the box. Each newly created `Box` is empty (the constructor should initialize `full` to false). Include `getter` and `setter` methods for all instance data. Include a `toString` method that returns a one-line description of the box. Create a driver class called `BoxTest`, whose main method instantiates and updates several `Box` objects.

PP B.4 Design and implement a class called `Book` that contains instance data for the title, author, publisher, and copyright date. Define the `Book` constructor to accept and initialize this data. Include setter and getter methods for all instance data. Include a `toString` method that returns a nicely formatted, multi-line description of the book. Create a driver class called `Bookshelf`, whose `main` method instantiates and updates several `Book` objects.

PP B.5 Design and implement a class called `Flight` that represents an airline flight. It should contain instance data that represents the airline name, flight number, and the flight's origin and destination cities. Define the `Flight` constructor to accept and initialize all instance data. Include getter and setter methods for all instance data. Include a `toString` method that returns a one-line description of the flight. Create a driver class called `FlightTest`, whose `main` method instantiates and updates several `Flight` objects.

PP B.6 Design a Java interface called `Priority` that includes two methods: `setPriority` and `getPriority`. The interface should define a way to establish numeric priority among a set of objects. Design and implement a class called `Task` that represents a task (such as on a to-do list) that implements the `Priority` interface. Create a driver class to exercise some `Task` objects.

PP B.7 Design a Java interface called `Lockable` that includes the following methods: `setKey`, `lock`, `unlock`, and `locked`. The `setKey`, `lock`, and `unlock` methods take an integer parameter that represents the key. The `setKey` method establishes the key. The `lock` and `unlock` methods lock and unlock the object, but only if the key passed in is correct. The `locked` method returns a boolean that indicates whether or not the object is locked. A `Lockable` object represents an object whose regular methods are protected: if the object is locked, the methods cannot be invoked; if it is unlocked, they can be invoked. Redesign and implement a version of the `Coin` class from Chapter 5 so that it is `Lockable`.

PP B.8 Design and implement a set of classes that define the employees of a hospital: doctor, nurse, administrator, surgeon, receptionist, janitor, and so on. Include methods in each class that are named according to the services provided by that person and that print an appropriate message. Create a `main` driver class to instantiate and exercise several of the classes.

PP B.9 Design and implement a set of classes that define various types of reading material: books, novels, magazines, technical journals, textbooks, and so on. Include data values that describe various attributes of the material, such as the number of pages and the names of the primary characters. Include methods that are named appropriately for each class and that print an appropriate message. Create a `main` driver class to instantiate and exercise several of the classes.

PP B.10 Design and implement a set of classes that keeps track of demographic information about a set of people, such as age, nationality, occupation, income, and so on. Design each class to focus on a particular aspect of data collection. Create a `main` driver class to instantiate and exercise several of the classes.

PP B.11 Design and implement a program that creates an exception class called `StringTooLongException`, designed to be thrown when a string is discovered that has too many characters in it. In the `main` driver of the program, read strings from the user until the user enters “DONE”. If a string is entered that has too many characters (say 20), throw the exception. Allow the thrown exception to terminate the program.

PP B.12 Modify the solution to Programming Project 10.1 such that it catches and handles the exception if it is thrown. Handle the exception by printing an appropriate message, and then continue processing more strings.

Answers to Self-Review Questions

SRA B.1 A class is the implementation of the blueprint for an object. An object is a specific instance of a class.

SRA B.2 Objects should be self-governing meaning that only the methods of a particular object should be able to access or modify the objects variables.

SRA B.3 Describe each of the following:

a. public method—is a method within a class that has public visibility and may be called by a method of any other class that has declared a variable of the first class.

b. private method—is a method that has private visibility and may only be accessed by methods within the class.

c. public variable—is a variable within a class that has public visibility and may be accessed by any method of any class that has declared a variable of the first class.

d. private variable—is a variable within a class that has private visibility and may only be accessed by methods within the class.

SRA B.4 A constructor is the method that is called in the creation of an instance of a class. The constructor will typically initialize object attributes. Multiple constructors may be provided for various initialization strategies (e.g., no parameters for a default initialization or one or more parameters for more specific initializations).

SRA B.5 Overloaded methods are distinguished from each other by their signatures including the number and type of the parameters.

SRA B.6 An aggregate object is an object that is made up of other objects.

SRA B.7 A static variable is shared among all instances of a class where as an instance variable is unique to a particular instance.

SRA B.8 A class provides implementations for all of its methods (unless it is an abstract class) where as an interface simply provides the headings for each method.

SRA B.9 The relationship between a child class and its parent class is called an is-a relationship. For example if class B is derived from class A, then B is an instance of A with whatever additional information and methods that are provided in B.

SRA B.10 Is-a.

SRA B.11 The java.lang.Object class is the root of the class hierarchy for the Java language. This means that all classes in Java are ultimately derived from the Object class.

SRA B.12 Polymorphism means having many forms. In object-oriented programming, we refer to an object reference as polymorphic if it can refer to objects of multiple classes.

SRA B.13 Overriding is related to polymorphism because a parent class reference can point to objects of any of its descendant classes. One or more of these classes may have overriden methods from the parent class causing the behavior of such a method to be dependent upon the type of the object referenced by the call.

SRA B.14 Polymorphism can be accomplished using interfaces because reference variables can be created using the interface type. Then those references can point to objects of any class that implements the interface.

Index