CHAPTER 7

Stacks

This chapter introduces a well-known ADT called a stack and presents both its applications and implementations. You will see how the operations on a stack give it a last-in, first-out behavior. Two of the several applications of a stack that the chapter considers are evaluating algebraic expressions and searching for a path between two points. Finally, the chapter discusses the important relationship between stacks and recursion.

7.1 The Abstract Data Type Stack

The specification of an abstract data type that you can use to solve a particular problem can emerge during the design of the problem's solution. The ADT developed in the following example happens to be an important one: the ADT stack.

Developing an ADT During the Design of a Solution

When you type a line of text on a keyboard, you are likely to make mistakes. If you use the backspace key to correct these mistakes, each backspace erases the previous character entered. Consecutive backspaces are applied in sequence and so erase several characters. For instance, if you type the line

```
abcc←ddde←←←ef←fg
```

where ← represents the backspace character, the corrected input would be

```
abcdefg
```

How can a program read the original line and get the correct input? In designing a solution to this problem, you eventually must decide how to store the input line. In accordance with the ADT approach, you should postpone this decision until you have a better idea of what operations you will need to perform on the data.

A first attempt at a solution leads to the following pseudocode:

Initial draft of a solution

```
// read the line, correcting mistakes along the way
while (not end of line) {
  Read a new character ch
  if (ch is not a '←') {
    Add ch to the ADT
  }
  else {
    Remove from the ADT the item added most recently
  }  // end if
}  // end while
```

This solution calls to attention two of the operations that the ADT should include:

Two ADT operations that are required

- Add a new item to the ADT.
- Remove from the ADT the item that was added most recently.

Notice that potential trouble lurks if you type ... empty ... when the ADT contains ... characters ... ensure ... two options: (1) ... (2) ... the program ...

```
// read the line, correcting mistakes along the way
while (not end of line) {
  Read a new character ch
  if (ch is not a '←') {
    Add ch to the ADT
  }
  else if (the ADT is not empty) {
    Remove from the ADT the item added most recently
  }
  else {
    Ignore the '←'
  } // end if
} // end while
```

The "read and correct" algorithm

From this pseudocode you can identify a third operation required by the ADT:

- Determine whether the ADT is empty.

Another required ADT operation

This solution places the corrected input line in the ADT. Now suppose that you want to display the line. At first, it appears that you can accomplish this task by using the ADT operations already identified, as follows:

```
// write the line
while (the ADT is not empty) {
  Remove from the ADT the item added most recently
  Write .....Uh-oh!
} // end while
```

A false start at writing the line

This pseudocode is incorrect for two reasons:

1. When you remove an item from the ADT, the item is gone, so you cannot write it. What you should have done was to *retrieve* from the ADT the item that was added most recently. A retrieval operation means to *look at, but leave unchanged*. Only after retrieving and writing the item should you remove it from the ADT.
2. The most recently added item is the last character of the input line. You certainly do not want to write it first. The resolution of this particular difficulty is left to you as an exercise.

Reasons why the attempted solution is incorrect

If we address only the first difficulty, the following pseudocode writes the input line in reversed order:

```
// write the line in reversed order
while (the ADT is not empty) {
  Retrieve from the ADT the item that was
    added most recently and put it in ch
  Write ch
  Remove from the ADT the item added most recently
} // end while
```

The write-backward algorithm

Thus, a fourth operation is required by the ADT:

Another required ADT operation

- Retrieve from the ADT the item that was added most recently.

Although you have yet to think about an implementation of the ADT, you know that you must be able to perform four specific operations.[1] These operations define the required ADT, which happens to be well known: It is usually called a **stack.** As you saw in Chapter 4, it is customary to include initialization operations in an ADT. Thus, the following operations define the ADT stack.

KEY CONCEPTS

ADT Stack Operations

1. Create an empty stack.
2. Determine whether a stack is empty.
3. Add a new item to the stack.
4. Remove from the stack the item that was added most recently.
5. Remove all the items from the stack.
6. Retrieve from the stack the item that was added most recently.

The term "stack" is intended to conjure up visions of things encountered in daily life, such as a stack of dishes in the school cafeteria, a stack of books on your desk, or a stack of assignments waiting for you to work on them. In common English usage, "stack of" and "pile of" are synonymous. To computer scientists, however, a stack is not just any old pile. A stack has the property that the last item placed on the stack will be the first item removed. This property is commonly referred to as **last-in, first-out,** or simply **LIFO.**

Last-in, first-out

A stack of dishes in a cafeteria makes a very good analogy of the abstract data type stack, as Figure 7-1 illustrates. As new dishes are added, the old dishes drop farther into the well beneath the surface. At any particular time, only the dish last placed on the stack is above the surface and visible. This dish is at the **top** of the stack and is the one that must be removed next. In general, the dishes are removed in exactly the opposite order from that in which they were added.

The LIFO property of stacks seems inherently unfair. Think of the poor person who finally gets the last dish on the cafeteria's stack, a dish that may have been placed there six years ago. Or how would you like to be the first person to arrive on the stack for a movie—as opposed to the line for a movie.

1. As you will learn if you complete Exercise 8 at the end of this chapter, the final algorithm to write the line correctly instead of in reversed order does not require additional ADT operations.

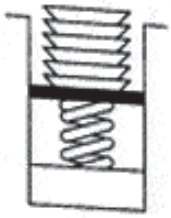


FIGURE 7-1
Stack of cafeteria dishes

You would be the last person allowed in! These examples demonstrate the reason that stacks are not especially prevalent in everyday life. The property that we usually desire in our daily lives is **first in, first out**, or **FIFO. A queue,** which you will study in the next chapter, is the abstract data type with the FIFO property. Most people would much prefer to wait in a movie *queue*—as a line is called in Britain—than in a movie *stack*. However, while the LIFO property of stacks is not appropriate for very many day-to-day situations, it is precisely what is needed for a large number of problems that arise in computer science.

Notice how well the analogy holds between the abstract data type stack and the stack of cafeteria dishes. The operations that manipulate data in the ADT stack are the *only* such operations, and they correspond to the only things that you can do to a stack of dishes. You can determine whether the stack of dishes is empty but not how many dishes are on the stack; you can inspect the top dish but no other dish; you can place a dish on top of the stack but at no other position; and you can remove a dish from the top of the stack but from no other position. If any of these operations was not available, or if you were permitted to perform any other operations, the ADT would not be a stack.

Although the stack of cafeteria dishes suggests that, as you add or remove dishes, the other dishes move, do not have this expectation of the ADT stack. The stack operations involve only the top item and imply only that the other items in the stack remain in sequence. Implementations of the ADT stack operations might or might not move the stack's items. The implementations given in this chapter do not move data items.

Refining the definition of the ADT stack. Before we specify the details of the stack operations, consider the removal and retrieval operations more carefully. The current definition enables you to remove the stack's top without inspecting it, or to inspect the stack's top without removing it. Both tasks are reasonable and occur in practice. However, if you wanted to inspect *and* remove the top item of a stack—a task that is not unusual—you would need the sequence of operations

- Retrieve from the stack the item that was added most recently.
- Remove from the stack the item that was added most recently.

[illegible]

[illegible] ADT stack in [illegible] generic. The [illegible] items are [illegible] the [illegible] specification of a module [illegible] ADT's [illegible] design [illegible] specification is design. For [illegible] the algorithms [illegible]

[illegible] the algorithms [illegible]

[illegible]
displays the input lines in reversed order by

KEY CONCEPTS

Pseudocode for the ADT Stack Operations

```
   StackItemType is the type of the items stored in the stack.

-createStack
   Creates an empty stack.

-isEmpty : boolean {query}
   Determines whether a stack is empty.

-push (in newItem:StackItemType) throws StackException
   Adds newItem to the top of the stack. Throws
   StackException if the insertion is not successful.

-pop : StackItemType throws StackException
   Retrieves and then removes the top of the stack -- the
   item that was added most recently . Throws
   StackException if the deletion is not successful.

-popAll;
// Removes all items from the stack.

-peek():StackItemType {query} throws StackException
// Retrieves the top of the stack. That is, peek
// retrieves the item that was added most recently.
// Retrieval does not change the stack. Throws
// StackException if the retrieval is not successful.
```

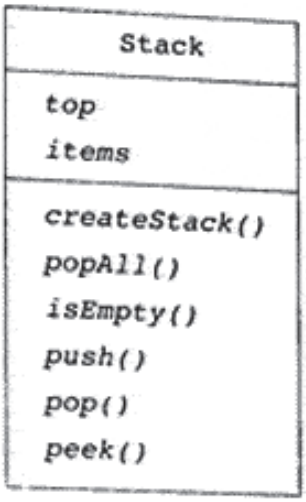


FIGURE 7-2

...agram for the class **Stack**

```
...iting the contents of stack.
    aStack = readAndCorrect()
while (!aStack.isEmpty()) {
  newChar = aStack.pop()
  Write newChar
  end while

Advance to new line


-readAndCorrect():Stack
  Reads the input line and returns the corrected
  version as a stack. For each character read,
  either enters it into the stack or, if it
  is '←', corrects the contents of stack.

aStack.createStack()[2]
Read newChar
while (newChar is not the end-of-line symbol) {
  if (newChar is not '←') {
    aStack.push(newChar)
  }
  else if (!aStack.isEmpty()) {
    oldChar = aStack.pop()
```

2 You implement the step **aStack.createStack()** in Java by declaring **aStack** as an instance of the stack class, since **createStack** is implemented as the class's constructor.

```
    } // end if
    Read newChar
  } // end while
  return aStack
```

We have used the stack operations without knowing their implementations or even what a stack looks like. Because the ADT approach builds a wall around the implementation of the stack, your program can use a stack independently of the stack's implementation. As long as the program correctly uses the ADT operations—that is, as long as it honors the contract—it will work regardless of how you implement the ADT.

The contract, therefore, must be written precisely. That is, before you implement any ADT operations, you should specify both their preconditions and their postconditions. Realize, however, that during program design, the first attempt at specification is often informal and is only later made precise by the writing of preconditions and postconditions.

Axioms (*optional*). As Chapter 4 noted, intuitive specifications, such as those given previously for the stack operations, are not really sufficient to define an ADT formally. For example, to capture formally the intuitive notion that the last item inserted into `aStack` is the first item to be removed, you could write an axiom such as

An example of an axiom

```
(aStack.push(newItem)).pop() = aStack
```

That is, if you push `newItem` onto `aStack` and then pop it, you are left with the original stack `aStack`. Exercise 16 at the end of this chapter discusses the axioms for a stack further.

7.2 Simple Applications of the ADT Stack

This section presents two rather simple examples for which the LIFO property of stacks is appropriate. Note that we will be using the operations of the ADT stack, even though we have not discussed their implementations yet.

Checking for Balanced Braces

Java uses curly braces, "{" and "}", to delimit groups of statements. For example, braces begin and end a method's body. If you treat a Java program as a string of characters, you can use a stack to verify that a program contains balanced braces. For example, the braces in the string

```
abc{defg{ijk}{l{mn}}op}qr
```

are balanced, while the braces in the string

```
abc{def}}{ghij{kl}m
```

are not balanced. You can check whether a string contains balanced braces by traversing it from left to right. As you move from left to right, you match each successive close brace "}" with the most recently encountered unmatched open brace "{"; that is, the "{" must be to the left of the current "}". The braces are balanced if

1. Each time you encounter a "}", it matches an already encountered "{"
2. When you reach the end of the string, you have matched each "{"

Requirements for balanced braces

The solution requires that you keep track of each unmatched "{" and discard one each time you encounter a "}". One way to perform this task is to push each "{" encountered onto a stack and pop one off each time you encounter a "}". Thus, a first-draft pseudocode solution is

Initial draft of a solution

```
while (not at the end of the string) {
  if (the next character is a '{') {
    aStack.push('{')
  }
  else if (the character is a '}') {
    openBrace = aStack.pop()
  }  // end if
}  // end while
```

Although this solution correctly keeps track of braces, missing from it are the checks that conditions 1 and 2 are met—that is, that the braces are indeed balanced. To verify condition 1 when a "}" is encountered, you must check to see whether the stack is empty before popping from it. If it is empty, you terminate the loop and report that the string is not balanced. To verify condition 2, you must check that the stack is empty when the end of the string is reached.

Thus, the pseudocode solution to check for balanced braces in *aString* becomes

A detailed pseudocode solution to check a string for balanced braces

```
aStack.createStack()
balancedSoFar = true
i = 0

while (balancedSoFar and i < length of aString) {
  ch = character at position i in aString
  ++i
  // push an open brace
  if (ch is '{') {
    aStack.push('{')
  }
  // close brace
  else if (ch is '}') {
    if (!aStack.isEmpty()) {
```

```
        openBrace = aStack.pop()    // pop a matching open brace
      }                             // no matching open brace
      else {
        balancedSoFar = false
      } // end if
    } // end if
    // ignore all characters other than braces
  } // end while

  if (balancedSoFar and aStack.isEmpty()) {
    aString has balanced braces
  }
  else {
    aString does not have balanced braces
  } // end if
```

Figure 7-3 shows the stacks that result when this algorithm is applied to several simple examples.

It may have occurred to you that a simpler solution to this problem is possible. You need only keep a count of the current number of unmatched open braces.[3] You need not actually store the open braces in a stack. However, the

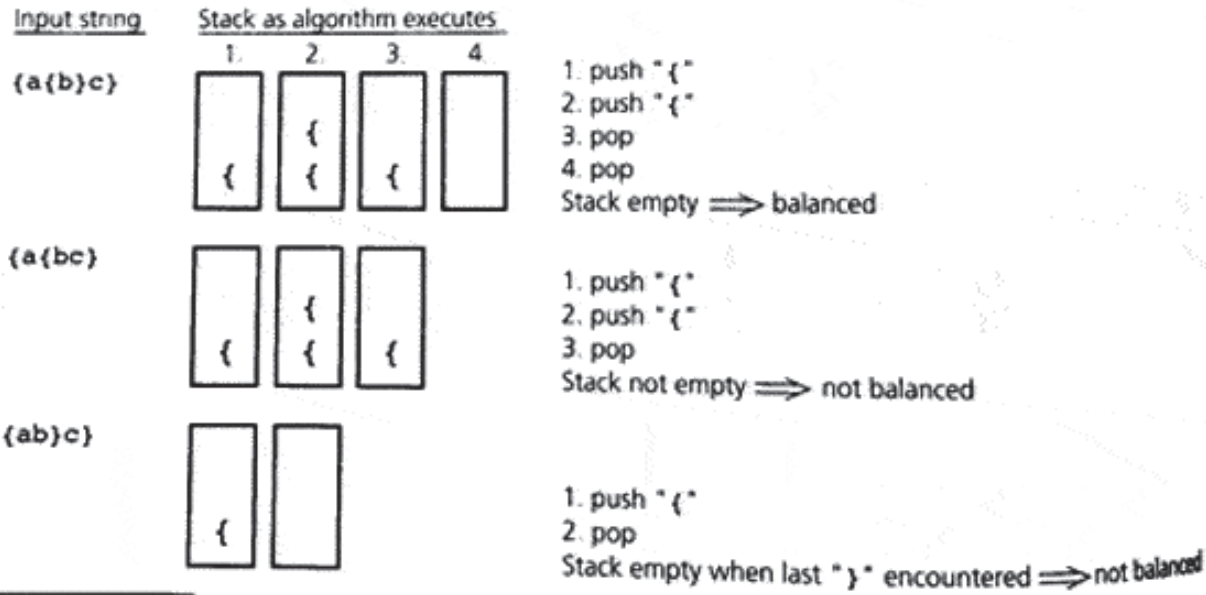


FIGURE 7-3

Traces of the algorithm that checks for balanced braces

3. Each time you encounter an open brace, you increment the count; each time you encounter a close brace, you decrement the count. If this count ever falls below zero or if it is greater than zero when the end of the string is reached, the string is unbalanced.

stack based solution is conceptually useful as it previews more legitimate uses of stacks. For example, Exercise 9 at the end of this chapter asks you to extend the algorithm given here to check for balanced parentheses and square brackets in addition to braces.

The exception `StackException`. Although the previous algorithm—in its present state of refinement—ignores the exception `StackException`, a Java implementation should not. The implementation either should take precautions to avoid an exception or should provide `try` and `catch` blocks to handle a possible exception. Suppose that `push` or `pop` throws `StackException`. Exactly how should you interpret this event?

The informality of the ADT specifications given earlier complicates the interpretation of `StackException`. For example, as it is specified, `StackException` is thrown if `pop` is not successful. Later, this chapter will clarify this particular specification: `pop` will be unsuccessful if it tries to delete an item from an empty stack. Under this assumption, you can refine the pseudocode

```
    close brace
else if (ch is '}') {
  if (!aStack.isEmpty()) {
    openBrace = aStack.pop()  // pop open brace
  }
  else {                      // no open brace
    balancedSoFar = false
  } // end if
} // end if
```

This section of the previous algorithm ignores **StackException**

in the previous algorithm to

```
// close brace
else if (ch is '}') {
  try {
    // try to pop open brace
    openBrace = aStack.pop()
  } // end try
  catch (StackException e) {
    balancedSoFar = false  // no open brace
  } // end catch
} // end if
```

Revision that makes use of **StackException**

The `push` operation can fail for implementation-dependent reasons. For example, `push` throws `StackException` if the array in an array-based implementation is full. In the spirit of fail-safe programming, a method that implements this balanced-braces algorithm should check for a thrown `StackException` after `push` and report an unsuccessful insertion.

Recognizing Strings in a Language

Consider the problem of recognizing whether a particular string is in the language

$$L = \{ w\$w' : w \text{ is a possibly empty string of characters other than \$, } w' = \text{reverse}(w) \}$$

For example, the strings AA, ABCCBA, and $ are in L, but AB$AB and ABC$CB are not. (Exercise 14 in Chapter 6 introduced a similar language.) This language is like the language of palindromes that you saw in Chapter 6, but strings in this language have a special middle character.

A stack is useful in determining whether a given string is in L. Suppose you traverse the first half of the string and push each character onto a stack. When you reach the $, you can undo the process: For each character in the second half of the string, you pop a character off the stack. However, you must match the popped character with the current character in the string to ensure that the second half of the string is the reverse of the first half. The stack must be empty when—and only when—you reach the end of the string; otherwise, one "half" of the string is longer than the other, and so the string is not in L.

The following algorithm uses this strategy. To avoid unnecessary complications, assume that *aString* contains exactly one $.

A pseudocode recognition algorithm for the language L

```
aStack.createStack()

// push the characters before $, that is, the
// characters in w, onto the stack
i = 0
ch = character at position i in aString
while (ch is not '$') {
  aStack.push(ch)
  ++i
  ch = character at position i in aString
}  // end while

// skip the $
++i

// match the reverse of w
inLanguage = true  // assume string is in language
while (inLanguage and i < length of aString) {
  ch = character at position i in aString
  try {
    stackTop = aStack.pop()
    if (stackTop equals ch) {
      ++i  // characters match
    }
    else {
```

```
      //  top of stack is not ch (characters do not match)
      inLanguage = false  // reject string
    }  // end if
  }  // end try
  catch (StackException e) {
    // aStack.pop() failed, aStack is empty (first half of
    // string is shorter than second half)
    inLanguage = false
  }  // end catch
}  // end while

if (inLanguage and aStack.isEmpty()) {
   aString is in language
}
else {
   aString is not in language
}  // end if
```

Notice that the two algorithms presented in this section depend only on the specifications of the stack operations and not on their implementations.

7.3 Implementations of the ADT Stack

This section develops three Java implementations of the ADT stack. The first implementation uses an array to represent the stack, the second uses a linked list, and the third uses the ADT list. Figure 7-4 illustrates these three implementations. The following interface *StackInterface* is used to provide a common specification for the three implementations.

```
public interface StackInterface {
  public boolean isEmpty();
  // Determines whether the stack is empty.
  // Precondition: None.
  // Postcondition: Returns true if the stack is empty;
  // otherwise returns false.

  public void popAll();
  // Removes all the items from the stack.
  // Precondition: None.
  // Postcondition: Stack is empty.

  public void push(Object newItem) throws StackException;
  // Adds an item to the top of a stack.
  // Precondition: newItem is the item to be added.
  // Postcondition: If insertion is successful, newItem
  // is on the top of the stack.
```

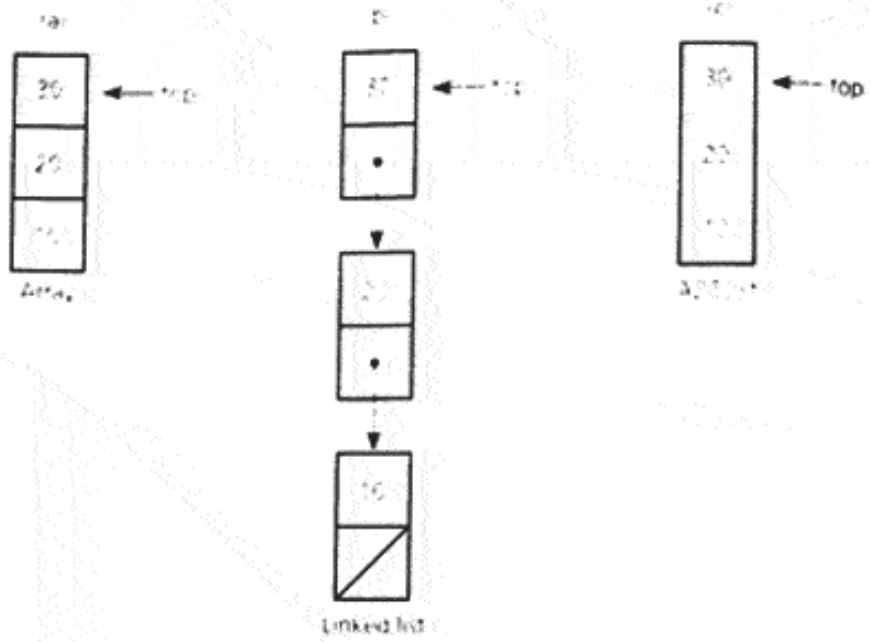


FIGURE 7-4

Implementations of the ADT stack that use (a) an array; (b) a linked list; (c) an ADT list

```
   // Exception: Some implementations may throw
   // StackException when newItem cannot be placed on
   // the stack.

   public Object pop() throws StackException;
   // Removes the top of a stack.
   // Precondition: None.
   // Postcondition: If the stack is not empty, the item
   // that was added most recently is removed from the
   // stack and returned.
   // Exception: Throws StackException if the stack is
   // empty.

   public Object peek() throws StackException;
   // Retrieves the top of a stack.
   // Precondition: None.
   // Postcondition: If the stack is not empty, the item
   // that was added most recently is returned. The
   // stack is unchanged.
   // Exception: Throws StackException if the stack is
   // empty.
} // end StackInterface
```

Here is the class *StackException* that is used in *StackInterface*:

```
public class StackException
             extends java.lang.RuntimeException {
  public StackException(String s) {
    super(s);
  }  // end constructor
}  // end StackException
```

Note that *StackException* extends *java.lang.RuntimeException*, so that the calls to methods that throw *StackException* do not have to be enclosed in *try* blocks. This is a reasonable choice for the operations *pop* and *peek*, since you can avoid the exception by checking to see whether the stack is empty before calling these operations. But *push* can also throw *StackException* in an array-based implementation when a fixed-size array is used and becomes full. You can avoid this exception in an array-based implementation by providing a method *isFull* that determines whether the stack is full; you call *isFull* before you call *push*.

In a reference-based implementation, this *isFull* method would not be necessary. Also, although the *push* method throws *StackException* in the interface specification, the *throws* clause could be omitted in a reference-based implementation of *push*.

An Array-Based Implementation of the ADT Stack

Figure 7-5 suggests that you use an array of *Objects* called *items* to represent the items in a stack and an index *top* such that *items[top]* is the stack's top. We want to define a class whose instances are stacks and whose private data fields are *items* and *top*.

The following class is an array-based implementation of the ADT stack. The default constructor for this class corresponds to and replaces the ADT operation *createStack*. Note that the preconditions and postconditions given earlier in *StackInterface* apply here as well, and so are omitted to save space.

```
public class StackArrayBased implements StackInterface {
  final int MAX_STACK = 50;  // maximum size of stack
  private Object items[];
  private int top;
```

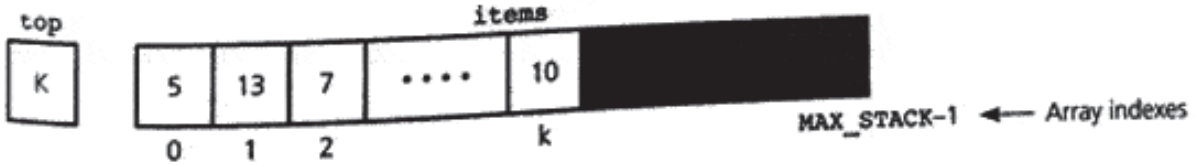


FIGURE 7-5

An array-based implementation

```
  public StackArrayBased() {
    items = new Object[MAX_STACK];
    top = -1;
  }  // end default constructor

  public boolean isEmpty() {
    return top < 0;
  }  // end isEmpty

  public boolean isFull() {
    return top == MAX_STACK-1;
  }  // end isFull

  public void push(Object newItem) throws StackException {
    if (!isFull()) {
      items[++top] = newItem;
    }
    else {
      throw new StackException("StackException on " +
                               "push: stack full");
    }  // end if
  }  // end push

  public void popAll() {
    items = new Object[MAX_STACK];
    top = -1;
  }  // end popAll

  public Object pop() throws StackException {
    if (!isEmpty()) {
      return items[top--];
    }
    else {
      throw new StackException("StackException on " +
                               "pop: stack empty");
    }  // end if
  }  // end pop

  public Object peek() throws StackException {
    if (!isEmpty()) {
      return items[top];
    }
    else {
      throw new StackException("Stack exception on " +
                               "peek - stack empty");
    }  // end if
```

```
  } // end peek
} // end StackArrayBased
```

A program that uses a stack could begin as follows:

```
public class StackTest {
  public static final int MAX_ITEMS = 15;

  public static void main(String[] args) {
    StackArrayBased stack = new StackArrayBased();
    Integer items[] = new Integer[MAX_ITEMS];
    for (int i=0; i<MAX_ITEMS; i++) {
      items[i] = new Integer(i);
      if (!stack.isFull()) {
        stack.push(items[i]);
      } // end if
    } // end for
    while (!stack.isEmpty()) {
      // cast result of pop to Integer
      System.out.println((Integer)(stack.pop()));
    } // end while
    ...
```

By implementing the stack as a class, and by declaring `items` and `top` as private, you ensure that the client cannot violate the ADT's walls. If you did not hide your implementation within a class, or if you made the array `items` public, the client could access the elements in `items` directly instead of by using the operations of the ADT stack. Thus, the client could access any elements in the stack, not just its top element. You might find this capability attractive, but in fact it violates the specifications of the ADT stack. If you truly need to access all the items of your ADT randomly, do not use a stack!

Private data fields are hidden from the client

Again, note that `StackException` provides a simple way for the implementer to indicate to the stack's client unusual circumstances, such as an attempted insertion into a full stack or a deletion from an empty stack.

`StackException` provides a simple way to indicate unusual events

Finally, note that instances of `StackArrayBased` cannot contain items of a primitive type such as `int`, because `int` is not derived from `Object`. If you need a stack of integers, for example, you will have to use the corresponding wrapper class, which in this case is `Integer`. Finally, `pop` and `peek` return an item that is an instance of `Object`. You must cast this item back to the subtype of `Object` that you pushed onto the stack. Otherwise, methods available for the subtype will not be accessible.

A Reference-Based Implementation of the ADT Stack

Many applications require a reference-based implementation of a stack so that the stack can grow and shrink dynamically. Figure 7-6 illustrates a reference-based

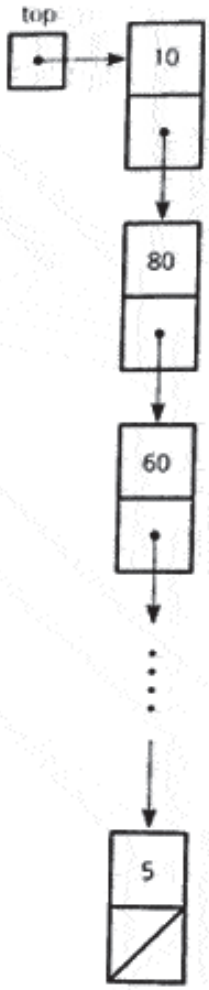


FIGURE 7-6

A reference-based implementation

implementation of a stack where *top* is a reference to the head of a linked list of items. The implementation uses the same node class developed for the linked list in Chapter 5.

Note that the preconditions and postconditions given earlier in *StackInterface* apply here as well, and so are omitted to save space.

```
public class StackReferenceBased
                    implements StackInterface {
 private Node top;

 public StackReferenceBased() {
   top = null;
 }  // end default constructor

 public boolean isEmpty() {
```

```
    return top == null;
  } // end isEmpty

  public void push(Object newItem) {
    top = new Node(newItem, top);
  } // end push

  public Object pop() throws StackException {
    if (!isEmpty()) {
      Node temp = top;
      top = top.next;
      return temp.item;
    }
    else {
      throw new StackException("StackException on " +
                               "pop: stack empty");
    } // end if
  } // end pop

  public void popAll() {
    top = null;
  } // end popAll

  public Object peek() throws StackException {
    if (!isEmpty()) {
      return top.item;
    }
    else {
      throw new StackException("StackException on " +
                               "peek: stack empty");
    } // end if
  } // end peek
} // end StackReferenceBased
```

An Implementation That Uses the ADT List

You can use the ADT list to represent the items in a stack, as Figure 7-7 illustrates. If the item in position 0 of a list represents the top of the stack, you can implement the stack operation *push(newItem)* as *add(0, newItem)*. Similarly, you can implement the stack operation *pop()* using *get(0)* and *remove(0)* and the stack operation *peek()* as *get(0)*.

Recall that Chapter 5 presented the ADT list as the class *ListReferenceBased*. (See page 265.) The following class for the ADT stack uses an instance of *ListReferenceBased* to represent the stack.

```
public class StackListBased implements StackInterface {
  private ListInterface list;
```

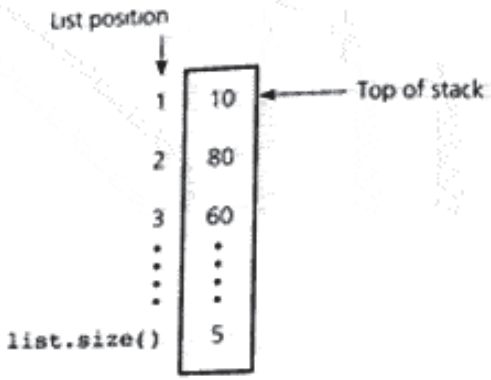


FIGURE 7-7

An implementation that uses the ADT list

```
public StackListBased() {
  list = new ListReferenceBased();
}  // end default constructor

public boolean isEmpty() {
  return list.isEmpty();
}  // end isEmpty

public void push(Object newItem) {
  list.add(0, newItem);
}  // end push

public Object pop() throws StackException {
  if (!list.isEmpty()) {
    Object temp = list.get(0);
    list.remove(0);
    return temp;
  }
  else {
    throw new StackException("StackException on " +
                             "pop: stack empty");
  }  // end if
}  // end pop

public void popAll() {
  list.removeAll();
}  // end popAll

public Object peek() throws StackException {
  if (!isEmpty()) {
    return list.get(0);
  }
```

```
    else {
      throw new StackException("StackException on " +
                               "peek: stack empty");
    }  // end if
  }  // end peek
}  // end StackListBased
```

The data field `list` is an instance of the class `ListReferenceBased`. Also, the class `ListReferenceBased`'s constructor is called by `StackList-Based`'s constructor.

Comparing Implementations

You have seen implementations of the ADT stack that use an array, a linked list, and the ADT list to represent the items in a stack. We have treated the array and linked list as data structures, but the list is an ADT that we have implemented by using either an array or a linked list. Thus, all our implementations of the ADT stack are ultimately array based or reference based.

Once again the reasons for making the choice between array-based and reference-based implementations are the same as those discussed in earlier chapters. The array-based implementation given in this chapter uses fixed-sized arrays. As such, it prevents the *push* operation from adding an item to the stack if the stack's size limit, which is the size of the array, has been reached. If this restriction is not acceptable, you must use either a resizeable array or a reference-based implementation. For the problem that reads and corrects an input line, for example, the fixed-size restriction might not present a difficulty: If the system allows a line length of only 80 characters, you could reasonably use a statically allocated array to represent the stack.

Fixed size versus dynamic size

Suppose that you decide to use a reference-based implementation. Should you choose the implementation that uses a linked list or the one that uses a reference-based implementation of the ADT list? Because a linked list actually represents the items on the ADT list, you might feel that using an ADT list to represent a stack is not as efficient as using a linked list directly. You would be right, but notice that the ADT list approach is much simpler to write. If you have battled references to produce a correct reference-based implementation of the ADT list, why do so again when you can *reuse* your work in the implementation of the stack? Which approach would you choose to produce a correct implementation of the stack in the least time? Chapter 9 discusses further the reuse of previously written classes.

Reuse of an already implemented class saves you time

The Java Collections Framework Class `Stack`

Chapter 5 introduced the Java Collections Framework (JCF) and the interface `List` that was used for the implementation of JCF list classes such as `LinkedList` and `ArrayList`. The JCF also contains an implementation of a stack class called `Stack`. Like many of the classes and interfaces we have seen so far from the JCF, the `Stack` class is a generic class.

The *Stack* class is derived from the class *Vector*—a growable array of objects. It extends the *Vector* class with five methods that allow for a LIFO stack of objects. Most of these methods are quite similar to the ones presented in this chapter: *push*, *pop*, *empty*, and *peek*. An additional method, called *search*, allows you to determine how far an item is from the top of the stack. Here is the specification for the JCF *Stack* collection as it is derived from *Vector*, only the method headings are shown:

```
public class Stack<E> extends Vector<E> {

  public Stack()
  // Creates an empty Stack

  public boolean empty()
  // Tests if this stack is empty.

  public E peek() throws EmptyStackException
  // Looks at the object at the top of this stack without
  // removing it from the stack.

  public E pop() throws EmptyStackException
  // Removes the object at the top of this stack and
  // returns that object as the value of this function.

  public E push(E item)
  // Pushes an item onto the top of this stack.

  public int search(Object o)
  // Returns the 1-based position where an object is on this
  // stack. The topmost item on the stack is considered to be
  // at distance 1.

} // end Stack
```

Note that the *Stack* has one data-type parameter for the items contained in the stack. Here is an example of how the JCF *Stack* is used:

```
import java.util.Stack;

public class TestStack {

  static public void main(String[] args) {
    Stack<Integer> aStack = new Stack<Integer>();
    if (aStack.empty()) {
      System.out.println("The stack is empty");
    } // end if
```

```
    for (int i = 0; i < 5; i++) {
      aStack.push(i); // With autoboxing, this is the same
                      // as aStack.push(new Integer(i))
    } // end for

    while (!aStack.empty()) {
      System.out.print(aStack.pop()+ " ");
    } // end while
    System.out.println();

  } // end main

} // end TestStack
```

The output of this program is

```
The stack is empty
4 3 2 1 0
```

7.4 Application: Algebraic Expressions

This section contains two more problems that you can solve neatly by using the ADT stack. Keep in mind throughout that you are using the ADT stack to solve the problems. You can use the stack operations, but you may not assume any particular implementation. You choose a specific implementation only as a last step.

Your use of an ADT's operations should not depend on its implementation

Chapter 6 presented recursive grammars that specified the syntax of algebraic expressions. Recall that prefix and postfix expressions avoid the ambiguity inherent in the evaluation of infix expressions. We will now consider stack-based solutions to the problems of evaluating infix and postfix expressions. To avoid distracting programming issues, we will allow only the binary operators *, /, +, and -, and disallow exponentiation and unary operators.

To evaluate an infix expression, first convert it to postfix form and then evaluate the postfix expression

The strategy we shall adopt here is first to develop an algorithm for evaluating postfix expressions and then to develop an algorithm for transforming an infix expression into an equivalent postfix expression. Taken together, these two algorithms provide a way to evaluate infix expressions. This strategy eliminates the need for an algorithm that directly evaluates infix expressions, a somewhat more difficult problem that Programming Problem 7 at the end of this chapter considers.

Evaluating Postfix Expressions

As we mentioned in Chapter 6, some calculators require you to enter postfix expressions. For example, to compute the value of

2 * (3 + 4)

by using a postfix calculator, you would enter the sequence 2, 3, 4, +, *, which corresponds to the postfix expression

2 3 4 + *

Recall that an operator in a postfix expression applies to the two operands that immediately precede it. Thus, the calculator must be able to retrieve the operands entered most recently. The ADT stack provides this capability. In fact, each time you enter an operand, the calculator pushes it onto a stack. When you enter an operator, the calculator applies it to the top two operands on the stack, pops the operands from the stack, and pushes the result of the operation onto the stack. Figure 7-8 shows the action of the calculator for the previous sequence of operands and operators. The final result, 14, is on the top of the stack.

You can formalize the action of the calculator to obtain an algorithm that evaluates a postfix expression, which is entered as a string of characters. To avoid issues that cloud the algorithm with programming details, assume that

- The string is a syntactically correct postfix expression
- No unary operators are present
- No exponentiation operators are present
- Operands are single lowercase letters that represent integer values

The pseudocode algorithm is then

```
for (each character ch in the string) {
  if (ch is an operand) {
    Push value that operand ch represents onto stack
  }
```

Expression	Calculator action		Stack (bottom to top)
2	push 2		2
3	push 3		2 3
4	push 4		2 3 4
+	operand2 = pop stack	(4,	2 3
	operand1 = pop stack	(3,	2
	result = operand1 + operand2	(7,	2
	push result		2 7
*	operand2 = pop stack	(7,	2
	operand1 = pop stack	(2,	
	result = operand1 * operand2	(14,	
	push result		14

FIGURE 7-8

The action of a postfix calculator when evaluating the expression 2 * (3 + 4)

```
    else {  // ch is an operator named op
      // evaluate and push the result
      operand2 = Pop the top of the stack
      operand1 = Pop the top of the stack
      result = operand1 op operand2
      Push result onto stack
    }  // end if
  }  // end for
```

Upon termination of the algorithm, the value of the expression will be on the top of the stack. Programming Problem 4 at the end of this chapter asks you to implement this algorithm.

Converting Infix Expressions to Equivalent Postfix Expressions

Now that you know how to evaluate a postfix expression, you will be able to evaluate an infix expression, if you first can convert it into an equivalent postfix expression. The infix expressions here are the familiar ones, such as $(a + b) * c / d - e$. They allow parentheses, operator precedence, and left-to-right association.

Will you ever want to evaluate an infix expression? Certainly, you have written such expressions in programs. The compiler that translated your programs had to generate machine instructions to evaluate the expressions. To do so, the compiler first transformed each infix expression into postfix form. Knowing how to convert an expression from infix to postfix notation not only will lead to an algorithm to evaluate infix expressions, but also will give you some insight into the compilation process.

If you manually convert a few infix expressions to postfix form, you will discover three important facts:

Facts about converting from infix to postfix

- The operands always stay in the same order with respect to one another.
- An operator will move only "to the right" with respect to the operands; that is, if, in the infix expression, the operand *x* precedes the operator *op*, it is also true that in the postfix expression, the operand *x* precedes the operator *op*.
- All parentheses are removed.

As a consequence of these three facts, the primary task of the conversion algorithm is determining where to place each operator.

The following pseudocode describes a first attempt at converting an infix expression to an equivalent postfix expression *postfixExp*:

First draft of an algorithm to convert an infix expression to postfix form

```
Initialize postfixExp to the null string
for (each character ch in the infix expression) {
  switch (ch) {
```

```
        case ch is an operand:
          Append ch to the end of postfixExp
          break
        case ch is an operator:
          Store ch until you know where to place it
          break
        case ch is '(' or ')':
          Discard ch
          break
      } // end switch
    } //  end for
```

You may have guessed that you really do not want to simply discard the parentheses, as they play an important role in determining the placement of the operators. In any infix expression, a set of matching parentheses defines an isolated subexpression that consists of an operator and its two operands. Therefore, the algorithm must evaluate the subexpression independently of the rest of the expression. Regardless of what the rest of the expression looks like, the operator within the subexpression belongs with the operands in that subexpression. The parentheses tell the rest of the expression

You can have the value of this subexpression after it is evaluated; simply ignore everything inside.

Parentheses, operator precedence, and left-to-right association determine where to place operators in the postfix expression

Parentheses are thus one of the factors that determine the placement of the operators in the postfix expression. The other factors are precedence and left-to-right association.

In Chapter 6, you saw a simple way to convert a fully parenthesized infix expression to postfix form. Because each operator corresponded to a pair of parentheses, you simply moved each operator to the position marked by its closing parenthesis, and finally removed the parentheses.

The actual problem is more difficult, however, because the infix expression is not always fully parenthesized. Instead, the problem allows precedence and left-to-right association, and therefore requires a more complex algorithm. The following is a high-level description of what you must do when you encounter each character as you read the infix string from left to right.

Five steps in the process to convert from infix to postfix form

1. When you encounter an operand, append it to the output string `postfixExp`. *Justification*: The order of the operands in the postfix expression is the same as the order in the infix expression, and the operands that appear to the left of an operator in the infix expression also appear to its left in the postfix expression.

2. Push each "(" onto the stack.

3. When you encounter an operator, if the stack is empty, push the operator onto the stack. However, if the stack is not empty, pop operators of greater or equal precedence from the stack and append them to `postfixExp`. Stop when you encounter either a "(" or an operator of lower precedence

ch	stack (bottom to top)	postfixExp	
a		a	
–	–	a	
(	– (	a	
b	– (	ab	
+	– (+	ab	
c	– (+	abc	
*	– (+ *	abc	
d	– (+ *	abcd	
)	– (+	abcd*	Move operators
	– (	abcd*+	from stack to
	–	abcd*+	postfixExp until " ("
/	– /	abcd*+	
e	– /	abcd*+e	Copy operators from
		abcd*+e/–	stack to postfixExp

FIGURE 7-9

A trace of the algorithm that converts the infix expression $a - (b + c * d)/e$ to postfix form

or when the stack becomes empty. You then push the new operator onto the stack. Thus, this step orders the operators by precedence and in accordance with left-to-right association. Notice that you continue popping from the stack until you encounter an operator of strictly lower precedence than the current operator in the infix expression. You do not stop on equality, because the left-to-right association rule says that in case of a tie in precedence, the leftmost operator is applied first—and this operator is the one that is already on the stack.

4. When you encounter a ")", pop operators off the stack and append them to the end of `postfixExp` until you encounter the matching "(". *Justification:* Within a pair of parentheses, precedence and left-to-right association determine the order of the operators, and Step 3 has already ordered the operators in accordance with these rules.

5. When you reach the end of the string, you append the remaining contents of the stack to `postfixExp`.

For example, Figure 7-9 traces the action of the algorithm on the infix expression $a - (b + c * d)/e$, assuming that the stack and the string `postfixExp` are initially empty. At the end of the algorithm, `postfixExp` contains the resulting postfix expression *abcd*+e/–*.

You can use the previous five-step description of the algorithm to develop a fairly concise pseudocode solution, which follows. The symbol + means concatenate (append), so `postfixExp + x` means concatenate the string currently in `postfixExp` and the character `x`—that is, follow the string in `postfixExp` with the character `x`. Both the stack `stack` and the postfix expression `postfixExp` are initially empty.

A pseudocode algorithm that converts an infix expression to postfix form

```
for (each character ch in the infix expression) {
  switch (ch) {
    case operand:  // append operand to end of postfixExp
      postfixExp = postfixExp + ch
      break
    case '(':      // save '(' on stack
      aStack.push(ch)
      break
    case ')':      // pop stack until matching '('
      while (top of stack is not '(') {
        postfixExp = postfixExp + aStack.pop()
      }  // end while
      openParen = aStack.pop() // remove the open parenthesis
      break
    case operator:              // process stack operators of
                                // greater precedence
      while ( !aStack.isEmpty() and
              top of stack is not '(' and
              precedence(ch) <= precedence(top of stack) ) {
        postfixExp = postfixExp + aStack.pop()
      }  // end while

      aStack.push(ch)  // save new operator
      break
  }  // end switch
}  // end for
// append to postfixExp the operators remaining in the stack
while (!aStack.isEmpty()) {
   postfixExp = postfixExp + aStack.pop()
}  // end while
```

Because this algorithm assumes that the given infix expression is syntactically correct, it can ignore the possibility of a `StackException` on `pop`. Programming Problem 6 at the end of this chapter asks you to remove this assumption. In doing so, you will find that you must provide `try` and `catch` blocks for the stack operations.

7.5 Application: A Search Problem

This final application of stacks will introduce you to a general [illegible] problem. In this particular problem, you must find a path from [illegible] to some location point. You will solve the problem [illegible] and then using recursion. The recursion [illegible] between [illegible] recursion [illegible]

you can focus on the issue at hand—the use of stacks during problem solving—we will simplify the problem: For each customer request, just indicate whether a sequence of HPAir flights exists from the origin city to the destination city. The more realistic problem of actually producing an itinerary—that is, the sequence of flights—is considered in Programming Problem 12 at the end of this chapter.

Determine whether HPAir flies from one city to another

Imagine three input text files that specify all of the flight information for the airline as follows:

- The names of the cities that HPAir serves
- Pairs of city names; each pair represents the origin and destination of one of HPAir's flights
- Pairs of city names; each pair represents a request to fly from some origin to some destination

The program should then produce output such as

```
Request is to fly from Providence to San Francisco.
HPAir flies from Providence to San Francisco.

Request is to fly from Philadelphia to Albuquerque.
Sorry. HPAir does not fly from Philadelphia to Albuquerque.

Request is to fly from Salt Lake City to Paris.
Sorry. HPAir does not serve Paris.
```

Representing the flight data. The flight map in Figure 7-10 represents the routes that HPAir flies. An arrow from city C_1 to city C_2 indicates a flight from C_1 to C_2. In this case C_2 is **adjacent** to C_1 and the path from C_1 to C_2 is called a **directed path.** Notice that if C_2 is adjacent to C_1, it does not follow that C_1 is adjacent to C_2. For example, in Figure 7-10, there is a flight from city R to city X, but not from city X to city R. As you will see in Chapter 14, the map in Figure 7-10 is called a **directed graph.**

C_2 is adjacent to C_1 if there is a directed path from C_1 to C_2

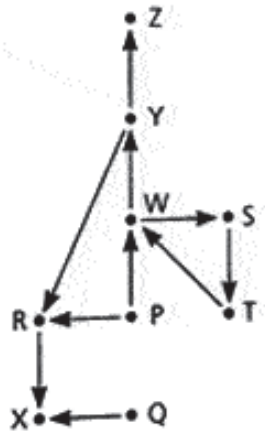


FIGURE 7-10

Flight map for HPAir

A Nonrecursive Solution That Uses a Stack

When processing a customer's request to fly from some origin city to some destination city, you must determine from the flight map whether there is a route from the origin to the destination. For example, by examining the flight map in Figure 7-10, you can see that a customer could fly from city P to city Z by flying first to city W, then to city Y, and finally to city Z; that is, there is a directed path from P to Z: $P \rightarrow W$, $W \rightarrow Y$, $Y \rightarrow Z$. Thus, you must develop an algorithm that searches the flight map for a directed path from the origin city to the destination city. Such a path might involve either a single flight or a sequence of flights. The solution developed here performs an **exhaustive search.** That is, beginning at the origin city, the solution will try every possible sequence of flights until either it finds a sequence that gets to the destination city or it determines that no such sequence exists. You will see that the ADT stack is useful in organizing this search.

Use a stack to organize an exhaustive search

First consider how you might perform the search by hand. One approach is to start at the origin city C_0 and select an arbitrary path to travel—that is, select an arbitrary flight departing from the origin city. This flight will lead you to a new city, C_1. If city C_1 happens to be the destination city, you are done; otherwise, you must attempt to get from C_1 to the destination city. To do this, you select a path to travel out of C_1. This path will lead you to a city C_2. If C_2 is the destination, you are done; otherwise, you must attempt to get from C_2 to the destination city, and so on.

Consider the possible outcomes of applying the previous strategy:

Possible outcomes of the exhaustive search strategy

1. You eventually reach the destination city and can conclude that it is possible to fly from the origin to the destination.
2. You reach a city C from which there are no departing flights.
3. You go around in circles. For example, from C_1 you go to C_2, from C_2 you go to C_3, and from C_3 you go back to C_1. You might continue this tour of the three cities forever; that is, the algorithm might enter an infinite loop.

If you always obtained the first outcome, everyone would be happy. However, because HPAir does not fly between all pairs of cities, you certainly cannot expect that the algorithm will always find a path from the origin city to the destination. For example, if city P in Figure 7-10 is the origin city and city Q is the destination city, the algorithm could not possibly find a path from city P to city Q.

Even if there were a sequence of flights from the origin city to the destination, it would take a bit of luck for the previous strategy to discover it—the algorithm would have to select a "correct" flight at each step. For example, even though there is a way to get from city P to city Z in Figure 7-10, the algorithm might not find it and instead might reach outcome 2 or 3. That is, suppose that from city P the algorithm chose to go to city R. From city R, the algorithm would have to go to city X, from which there are no flights out (outcome 2). On the other hand, suppose that the algorithm chose to go to city W

from city *P*. From city *W*, the algorithm might choose to go to city *S*. It would then have to go to city *T* and then back to *W*. From *W* it might once again choose to go to city *S* and continue to go around in circles (outcome 3).

You thus need to make the algorithm more sophisticated, so that it always finds a path from the origin to the destination, if such a path exists, and otherwise terminates with the conclusion that there is no such path. Suppose that the earlier strategy results in outcome 2: You reach a city *C* from which there are no departing flights. This certainly does not imply that there is no way to get from the origin to the destination; it implies only that there is no way to get from city *C* to the destination. In other words, it was a mistake to go to city *C*. After discovering such a mistake, the algorithm can retrace its steps, or *backtrack*, to the city *C'* that was visited just before city *C* was visited. Once back at city *C'*, the algorithm can select a flight to some city other than *C*. Notice that it is possible that there are no other flights out of city *C'*. If this were the case, it would mean that it was a mistake to visit city *C'*, and thus you would want to backtrack again, this time to the city that was visited just before city *C'*.

Use backtracking to recover from a wrong choice

For example, you saw that, in trying to get from city *P* to city *Z* in Figure 7-10, the algorithm might first choose to go from city *P* to city *R* and then on to city *X*. As there are no departing flights from city *X*, the algorithm must backtrack to city *R*, the city visited before city *X*. Once back at city *R*, the algorithm would attempt to go to some city other than city *X*, but would discover that this is not possible. The algorithm would thus backtrack once more, this time to city *P*, which was visited just before city *R*. From city *P*, the algorithm would choose to go to city *W*, which is a step in the right direction!

For the algorithm to implement this new strategy, it must maintain information about the order in which it visits the cities. First notice that when the algorithm backtracks from a city *C*, it must retreat to the city that it visited most recently before *C*. This observation suggests that you maintain the sequence of visited cities in a stack. That is, each time you decide to visit a city, you push its name onto the stack, as parts *a*, *b*, and *c* of Figure 7-11 illustrate for the flights from *P* to *R* to *X* in the previous example. You select the next city to visit from those adjacent to the city on the top of the stack. When you need to backtrack from the city *C* at the top of the stack (for example, because there are no flights out of the city), you simply pop a city from the stack, as shown in Figure 7-11d. After the pop, the city on the top of the stack is the city on the current path that you visited most recently before *C*. Parts *e* and *f* of Figure 7-11 illustrate the backtrack to city *P* and the subsequent flight to *W*.

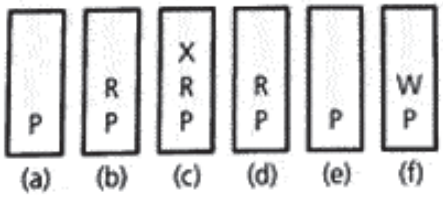


FIGURE 7-11

The stack of cities as you travel (a) from *P*; (b) to *R*; (c) to *X*; (d) back to *R*; (e) back to *P*; (f) to *W*

The algorithm, as developed so far, is

First draft of a search algorithm with backtracking

```
aStack.createStack()

aStack.push(originCity)  // push origin city onto stack

while (a sequence of flights from the origin to the
                              destination has not been found){
  if (you need to backtrack from the city on the
        top of the stack) {
     temp = aStack.pop()
  }
  else {
    Select a destination city C for a flight from
    the city on the top of the stack
    aStack.push(C)
  }  // end if
}  // end while
```

Notice that at any point in the algorithm, the contents of the stack correspond to the sequence of flights currently under consideration. The city on the top of the stack is the city you are visiting currently, directly "below" it is the city visited previously, and so forth down to the bottom city, which is the first city visited in the sequence, or the origin city. In other words, an invariant of the *while* loop is that

invariant

The stack contains a directed path from the origin city at the bottom of the stack to the city at the top of the stack.

You can therefore always retrace your steps as far back through the sequence as needed.

Now consider the question of when to

```
Backtrack from the city on the top of the stack.
```

You have already seen one case when backtracking is necessary. You must backtrack from the city on the top of the stack when there are no flights out of that city. Another time when you need to backtrack is related to the problem of going around in circles, described previously as the third possible outcome of the original strategy.

Backtrack when there are no more unvisited cities

A key observation that will tell you when to backtrack is, ***you never want to visit a city that the search has already visited.*** As a consequence, you must backtrack from a city whenever there are no more unvisited cities to fly to. To see why you never want to visit a city a second time, consider two cases:

Two reasons for not visiting a city more than once

- If you have visited city C and it is still somewhere in the stack—that is, it is part of the sequence of cities that you are exploring currently—you do not want to visit C again. Any sequence that goes from C through C_1, C_2, ..., C_k, back to C, and then to C' might just as well skip the intermediate cities and go from C directly to C'.

For example, suppose that the algorithm starts at P in Figure 7-10 and, in trying to find a path to Y, visits W, S, and T. There is now no reason for the algorithm to consider the flight from T to W because W is already in the stack. Anywhere you could fly to by going from W to S, from S to T, and then back to W, such as city Y, you could fly to directly from W without first going through S and T. Because you do not allow the algorithm to visit W a second time, it will backtrack from S and T to W and then go from W directly to Y. Figure 7-12 shows how the stack would appear if revisits were allowed and how it looks after backtracking when revisits are not allowed. Notice that backtracking to W is very different from visiting W for a second time.

- If you have visited city C, but it is no longer in the stack—because you backtracked from it and popped it from the stack—you do not want to visit C again. This situation is subtle; consider two cases that depend on why you backtracked from the city.

  If you backtracked from C because there were no flights out of it, then you certainly do not ever want to try going through C again. For example, if, starting at P in Figure 7-10, the algorithm goes to R and then to X, it will backtrack from X to R. At this point, although X is no longer in the stack, you certainly do not want to visit it again, because you know there are no flights out of X.

  Now suppose that you backtracked from city C because all cities adjacent to it had been visited. This situation implies that you have already tried all possible flights from C and have failed to find a way to get to the destination city. There is thus no reason to go to C again. For example, suppose that starting from P in Figure 7-10, the algorithm executes the following sequence: Visit R, visit X, backtrack to R (because there are no flights out of X), backtrack to P (because there are no more unvisited cities

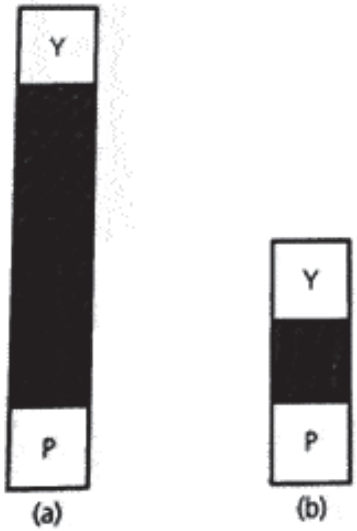


FIGURE 7-12

The stack of cities (a) allowing revisits and (b) after backtracking when revisits are not allowed

adjacent to R), visit W, visit Y. At this point the stack contains P-W-Y, with Y on top, as Figure 7-12b shows. You need to choose a flight out of Y. You do not want to fly from Y to R, because you have visited R already and tried all possible flights out of R.

In both cases, visiting a city a second time does not gain you anything, and in fact it may cause you to go around in circles.

Mark the visited cities

To implement the rule of not visiting a city more than once, you simply mark a city when it has been visited. When choosing the next city to visit, you restrict consideration to unmarked cities adjacent to the city on the top of the stack. The algorithm thus becomes

Next draft of the search algorithm

```
aStack.createStack()
Clear marks on all cities
aStack.push(originCity)  // push origin city onto stack
Mark the origin as visited
while (a sequence of flights from the origin to the
                          destination has not been found) {
   // loop invariant: The stack contains a directed path
   // from the origin city at the bottom of the stack to
   // the city at the top of the stack
   if (no flights exist from the city on the
         top of the stack to unvisited cities) {
     temp = aStack.pop()  // backtrack
   }
   else {
      Select an unvisited destination city C for a
        flight from the city on the top of the stack
      aStack.push(C)
      Mark C as visited
   }  // end if
}  // end while
```

Finally, you need to refine the condition in the `while` statement. That is, you need to refine the algorithm's final determination of whether a path exists from the origin to the destination. The loop invariant, which states that the stack contains a directed path from the origin city to the city on the top of the stack, implies that the algorithm can reach an affirmative conclusion if the city at the top of the stack is the destination city. On the other hand, the algorithm can reach a negative conclusion only after it has exhausted all possibilities—that is, after the algorithm has backtracked to the origin and there remain no unvisited cities to fly to from the origin. At that point, the algorithm pops the origin city from the stack and the stack will become empty.

With this refinement, the algorithm appears as follows:

```
// originCity to destinationCity

aStack.createStack()
Clear marks on all cities

aStack.push(originCity)  // push origin onto stack
Mark the origin as visited

while (!aStack.isEmpty() and
       destinationCity is not at the top of the stack) {
  // Loop invariant: The stack contains a directed path
  // from the origin city at the bottom of the stack to
  // the city at the top of the stack
  if (no flights exist from the city on the
          top of the stack to unvisited cities) {
    temp = aStack.pop()  // backtrack
  }
  else {
    Select an unvisited destination city C for a
        flight from the city on the top of the stack
    aStack.push(C)
    Mark C as visited
  }  // end if
}  // end while

if (aStack.isEmpty()) {
  return false  // no path exists
}
else {
  return true   // path exists
}  // end if
```

Notice that the algorithm does not specify the order of selection for the unvisited cities. It really does not matter what selection criteria the algorithm uses, because the choice will not affect the final outcome: Either a sequence of flights exists or it does not. The choice, however, will affect the specific flights that the algorithm considers. For example, suppose that the algorithm always flies to the alphabetically earliest unvisited city from the city on the top of the stack. Under this assumption, Figure 7-13 contains a trace of the algorithm's action, given the map in Figure 7-10, with *P* as the origin city and *Z* as the destination city. The algorithm terminates with success.

Now consider the operations that the search algorithm must perform on the flight map. The algorithm marks cities as it visits them, determines whether a city has been visited, and determines which cities are adjacent to a given city. You can treat the flight map as an ADT that has at least these operations, in addition to the search operation itself. Other desirable operations include placing data into the flight map, inserting a city adjacent to another city,

Action	Reason	Contents of stack (bottom to top)
Push P	Initialize	P
Push R	Next unvisited adjacent city	P R
Push X	Next unvisited adjacent city	P R X
Pop X	No unvisited adjacent city	P R
Pop R	No unvisited adjacent city	P
Push W	Next unvisited adjacent city	P W
Push S	Next unvisited adjacent city	P W S
Push T	Next unvisited adjacent city	P W S T
Pop T	No unvisited adjacent city	P W S
Pop S	No unvisited adjacent city	P W
Push Y	Next unvisited adjacent city	P W Y
Push Z	Next unvisited adjacent city	P W Y Z

FIGURE 7-13

A trace of the search algorithm, given the flight map in Figure 7-10

displaying the flight map, displaying a list of all cities, and displaying all cities that are adjacent to a given city. Thus, the ADT flight map could include the following operations:

ADT flight map operations

```
+createFlightMap()
// Creates an empty flight map.

+readFlightMap(in cityFileName:string,
               in flightFileName:string)
// Reads flight information into the flight map.

+displayFlightMap() {query}
// Displays flight information.

+displayAllCities() {query}
// Displays the names of all cities that HPAir serves.

+displayAdjacentCities(in aCity:City) {query}
// Displays all cities that are adjacent to a given city.

+markVisited(in aCity:City)
// Marks a city as visited.

+unvisitAll()
// Clears marks on all cities.

+isV[illegible](in aCity:City):boo[illegible] {query}
// D[illegible]s whe[illegible] city [illegible]visited

+[illegible]:City
```

```
+getNextCity(in fromCity:City)
// Returns the next unvisited city, if any, that
// is adjacent to a given city.  Returns null if no
// unvisited adjacent city was found.

+isPath(in originCity:City, in destinationCity:City)
// Determines whether a sequence of flights between
// two cities exists.
```

The following Java method implements the *isPath* operation by using the *searchS* algorithm. It assumes that the class *StackReferenceBased* implements the stack operations and the class *Map* implements the ADT flight map operations just described. Notice that you must represent the cities by creating a class *City* that implements the *java.lang.Comparable* interface.

Java implementation of **searchS**

```
public boolean isPath(City originCity,
                      City destinationCity) {
// ------------------------------------------------
// Determines whether a sequence of flights between two cities
// exists. Nonrecursive stack version.
// Precondition: originCity and destinationCity are the origin
// and destination cities, respectively.
// Postcondition: Returns true if a sequence of flights exists
// from originCity to destinationCity, otherwise returns
// false. Cities visited during the search are marked as
// visited in the flight map.
// Implementation notes: Uses a stack for the cities of a
// potential path. Calls unvisitAll, markVisited, and
// getNextCity.
// ------------------------------------------------
 StackReferenceBased stack = new StackReferenceBased();

 City topCity, nextCity;
 unvisitAll();  // clear marks on all cities

 // push origin city onto stack, mark it visited
 stack.push(originCity);
 markVisited(originCity);

 topCity = (City)(stack.peek());
 while (!stack.isEmpty() &&
        (topCity.compareTo(destinationCity) != 0)) {
   // loop invariant: stack contains a directed path
   // from the origin city at the bottom of the stack
   // to the city at the top of the stack

   // find an unvisited city adjacent to the city on
   // the top of the stack
```

```
      nextCity = getNextCity(topCity);

      if (nextCity == null) {
        stack.pop();  // no city found; backtrack
      }
      else {                        // visit city
        stack.push(nextCity);
        markVisited(nextCity);
      } // end if
      topCity = (City)stack.peek();
    } // end while
    if (stack.isEmpty()) {
      return false;  // no path exists
    }
    else {
      return true;   // path exists
    } // end if
  } // end isPath
```

Programming Problem 10 at the end of this chapter provides implementation details that will enable you to complete the solution to the HPAir problem.

A Recursive Solution

Recall the initial attempt at a solution to the HPAir problem of searching for a sequence of flights from some origin city to some destination city. Consider how you might perform the search "by hand." One approach is to start at the origin city and select an arbitrary flight that departs from the origin city. This flight will lead you to a new city, C_1. If city C_1 happens to be the destination city, you are done; otherwise, you must attempt to get from C_1 to the destination city by selecting a flight out of C_1. This flight will lead you to city C_2. If C_2 is the destination, you are done; otherwise, you must attempt to get from C_2 to the destination city, and so on. There is a distinct recursive flavor to this search strategy, which can be restated as follows:

A recursive search strategy

```
To fly from the origin to the destination:
  Select a city C adjacent to the origin
  Fly from the origin to city C
  if (C is the destination city) {
    Terminate -- the destination is reached
  }
  else {
    Fly from city C to the destination
  } // end if
```

This statement of the search strategy makes its recursive nature very appar
The first step in flying from the origin city to the destination city is to fly fr

the origin city to city C. Once at city C, you are confronted with another problem of the same type—you now must fly from city C to the destination.

This recursive formulation is nothing more than a restatement of the initial incomplete strategy developed previously. As such it has the same three possible outcomes:

Possible outcomes of the recursive search strategy

1. You eventually reach the destination city and can conclude that it is possible to fly from the origin to the destination.
2. You reach a city C from which there are no departing flights.
3. You go around in circles.

The first of these outcomes corresponds to a base case of the recursive algorithm. If you ever reach the destination city, no additional problems of the form "fly from city C to the destination" are generated, and the algorithm terminates. However, as was observed previously, the algorithm might not produce this outcome; that is, it might not reach this base case. The algorithm might reach a city C that has no departing flights. (Notice that the algorithm does not specify what to do in this case—in this sense the algorithm is incomplete. Or the algorithm might repeatedly cycle through the same sequence of cities and thus never terminate.

You can resolve these problems by mirroring what you did in the previous section. Consider the following refinement, in which you mark visited cities and never fly to a city that has been visited already:

A refinement of the recursive search algorithm

```
searchR(in originCity:City, in destinationCity:City):boolean
  Searches for a sequence of flights from
  originCity to destinationCity.

  Mark originCity as visited

  if (originCity is destinationCity) {
    Terminate -- the destination is reached
  }
  else {
    for (each unvisited city C adjacent to originCity) {
      searchR(C, destinationCity)
    }  // end for
  }  // end if
```

Now consider what happens when the algorithm reaches a city that has no unvisited city adjacent to it. For example, consider the piece of a flight map in Figure 7-14. When `searchR` reaches city *X*—that is, when the parameter `originCity` has the value *X*—the `for` loop will not be entered, because no unvisited cities are adjacent to *X*. Hence, the method `searchR` returns. This return has the effect of backtracking to city *W*, from which the flight to *X* originated. In terms of the previous pseudocode, the return is made to the point from which the call `searchR(X, destinationCity)` occurred. This point is within

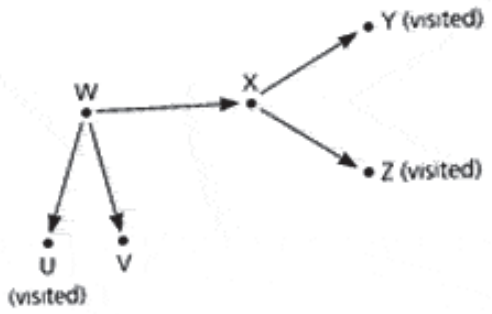


FIGURE 7-14

A piece of a flight map

the *for* loop, which iterates through the unvisited cities adjacent to *W;* that is, the parameter *originCity* has the value *W.*

After backtracking from *X* to *W,* the *for* loop will again execute. This time the loop chooses city *V,* resulting in the recursive call *searchR(V, destinationCity)*. From this point, the algorithm either will eventually reach the destination city and terminate, or it will backtrack once again to city *W.* If it backtracks to *W,* the *for* loop will terminate because there are no more unvisited cities adjacent to *W,* and a return from *searchR* will occur. The effect is to backtrack to the city where the flight to *W* originated. If the algorithm ever backtracks to the origin city and no remaining unvisited cities are adjacent to it, the algorithm will terminate, and you can conclude that no sequence of flights from the origin to the destination exists. Notice that the algorithm will always terminate in one way or another, because it will either reach the destination city or run out of unvisited cities to try.

The following Java method implements the *searchR* algorithm:

Java implementation of **searchR**

```
public boolean isPath(City originCity,
                      City destinationCity) {
  City  nextCity;
  boolean done;

  // mark the current city as visited
  markVisited(originCity);

  // base case: the destination is reached
  if (originCity.compareTo(destinationCity) == 0) {
    return true;
  }
  else { // try a flight to each unvisited city
    done = false;
    nextCity = getNextCity(originCity);

    while (nextCity != null && !done) {
      done = isPath(nextCity, destinationCity);
```

```
    if (!done) {
      nextCity = getNextCity(originCity);
    } // end if
  } // end while

  return done;
} // end if
} // end isPath
```

You have probably noticed a close parallel between this recursive algorithm and the earlier stack-based algorithm `searchS`. In fact, the two algorithms simply employ different techniques to implement the identical search strategy. The next section will elaborate on the relationship between the two algorithms.

7.6 The Relationship Between Stacks and Recursion

The previous section solved the HPAir problem once by using the ADT stack and again by using recursion. The goal of this section is to relate the way that the stack organizes the search for a sequence of flights to the way a recursive algorithm organizes the search. You will see that the ADT stack has a hidden presence in the concept of recursion and, in fact, that stacks have an active role in most computer implementations of recursion.

Consider how the two search algorithms implement three key aspects of their common strategy.

A comparison of key aspects of two search algorithms

- **Visiting a new city.** The recursive algorithm `searchR` visits a new city C by calling `searchR(C, destinationCity)`. The algorithm `searchS` visits city C by pushing C onto a stack. Notice that if you were to use the box trace to trace the execution of `searchR`, the call `searchR(C, destinationCity)` would generate a box in which the city C is associated with the formal parameter `originCity` of `searchR`.

  For example, Figure 7-15 shows both the state of the box trace for `searchR` and the stack for `searchS` at corresponding points of their search for a path from city P to city Z in Figure 7-10.

- **Backtracking.** Both search algorithms attempt to visit an unvisited city that is adjacent to the current city. Notice that this current city is the value associated with the formal parameter `originCity` in the deepest (rightmost) box of `searchR`'s box trace. Similarly, the current city is on the top of `searchS`'s stack. In Figure 7-15, this current city is X. If no unvisited cities are adjacent to the current city, the algorithms must backtrack to the previous city. The algorithm `searchR` backtracks by returning from the current recursive call. You represent this action in the box trace by crossing off the deepest box. The algorithm `searchS` backtracks by explicitly popping from its stack. For example, from the state depicted in Figure 7-15, both algorithms backtrack to city R and then to city P, as Figure 7-16 illustrates.

- **Termination.** The search algorithms terminate either when they reach the destination city or when they exhaust all possibilities. All possibilities are exhausted when, after backtracking to the origin city, no unvisited adjacent cities remain. This situation occurs for `searchR` when all boxes have been crossed off in the box trace and a return occurs to the point of the original call to the method. For `searchS`, no unvisited cities are adjacent to the origin when the stack becomes empty.

Thus, the two search algorithms really do perform the identical action. In fact, provided that they use the same rule to select an unvisited city—for example, traverse the current city's list of adjacent cities alphabetically—they will always visit the identical cities in the identical order. The similarities between the algorithms are far more than coincidence. In fact, it is always possible to capture the actions of a recursive method by using a stack.

Typically, stacks are used to implement recursive methods

An important context in which the close tie between stacks and recursion is explicitly utilized is a compiler's implementation of a recursive method. It is common for a compiler to use a stack to implement a recursive method in a manner that greatly resembles the box trace. When a recursive call to a method occurs, the implementation must remember certain information. This information consists essentially of the same local environment that you place in the boxes—values of both parameters and local variables, and a reference to the point from which the recursive call was made.

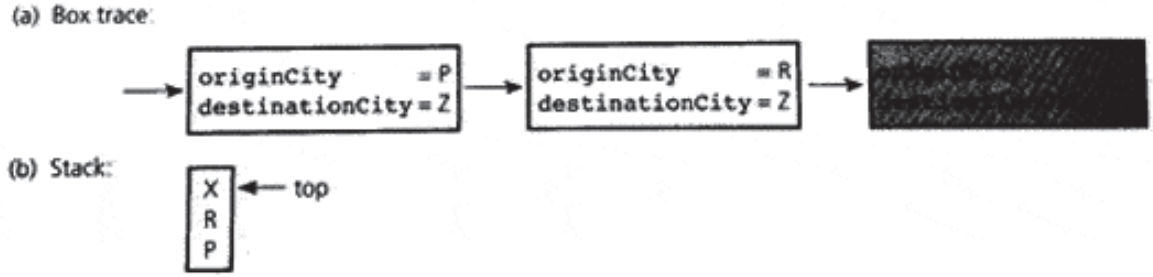


FIGURE 7-15

Visiting city *P*, then *R*, then *X*: (a) box trace versus (b) stack

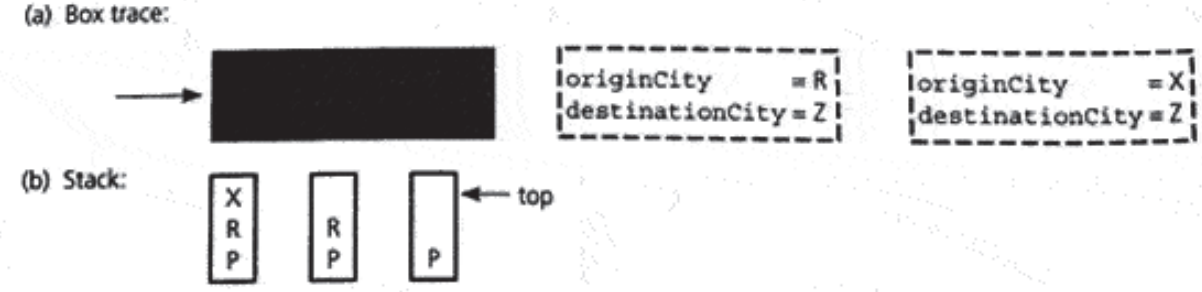


FIGURE 7-16

Backtracking from city *X* to *R* to *P*: (a) box trace versus (b) stack

During execution, the compiled program must manage these boxes of information, or **activation records**, just as you must manage them on paper. As the HPAir example has indicated, the operations needed to manage the activation records are those that a stack provides. When a recursive call occurs, a new activation record is created and pushed onto a stack. This action corresponds to the creation of a new box at the deepest point in the sequence. When a return is made from a recursive call, the stack is popped, bringing the activation record that contains the appropriate local environment to the top of the stack. This action corresponds to crossing off the deepest box and following the arrow back to the preceding box. Although we have greatly simplified the process, most implementations of recursion are based on stacks of activation records.

Each recursive call generates an activation record that is pushed onto a stack

You can use a similar strategy to implement a nonrecursive version of a recursive algorithm. You might need to recast a recursive algorithm into a nonrecursive form to make it more efficient, as mentioned in Chapter 3. The previous discussion should give you a taste of the techniques for removing recursion from a program. You will encounter recursion removal as a formal topic in more advanced courses, such as compiler construction.

You can use stacks when implementing a nonrecursive version of a recursive algorithm

Summary

1. The ADT stack operations have a last-in, first-out (LIFO) behavior.
2. Algorithms that operate on algebraic expressions are an important application of stacks. The LIFO nature of stacks is exactly what the algorithm that evaluates postfix expressions needs to organize the operands. Similarly, the algorithm that transforms infix expressions to postfix form uses a stack to organize the operators in accordance with precedence rules and left-to-right association.
3. You can use a stack to determine whether a sequence of flights exists between two cities. The stack keeps track of the sequence of visited cities and enables the search algorithm to backtrack easily. However, displaying the sequence of cities in their normal order from origin to destination is awkward, because the origin city is at the bottom of the stack and the destination is at the top.
4. A strong relationship between recursion and stacks exists. Most implementations of recursion maintain a stack of activation records in a manner that resembles the box trace.

Cautions

1. Operations such as *peek* and *pop* must take reasonable action when the stack is empty. One possibility is to ignore the operation and throw an exception `StackException`.
2. Algorithms that evaluate an infix expression or transform one to postfix form must determine which operands apply to a given operator. Doing so allows for precedence and left-to-right association so that you can omit parentheses.
3. When searching for a sequence of flights between cities, you must take into account the possibility that the algorithm will make wrong choices. For example,

the algorithm must be able to backtrack when it hits a dead end, and you must eliminate the possibility that the algorithm will cycle.

Self-Test Exercises

1. If you push the letters W, Y, X, Z, and V in order onto a stack of characters and then pop them, in what order will they be deleted from the stack?

2. What do the initially empty stacks `stack1` and `stack2` "look like" after the following sequence of operations?

```
stack1.push(23)
stack1.push(17)
stack1.push(50)
stack2.push(42)
top1 = stack1.pop()
top2 = stack2.peek()
stack2.push(top1)
stack1.push(top2)
stack1.push(13)
top2 = stack2.pop()
stack2.push(49)
```

   Compare these results with Self-Test Exercise 2.

3. The algorithms that appear in the section "Simple Applications of the ADT Stack" involve strings. Under what conditions would you choose an array-based implementation for the stack in these algorithms? Under what conditions would you choose a reference-based implementation?

4. Describe the difference between the *peek* operation and the *pop* operations in an ADT stack.

5. For each of the following strings, trace the execution of the balanced-braces algorithm and show the contents of the stack at each step.

   a. `x{{{y}z}`

   b. `{x{y{z}}}`

   c. `{xy{z}}}`

6. Use the stack algorithms in this chapter to evaluate the postfix expression $ab-c+$. Assume the following values for the identifiers: $a = 7$; $b = 3$; $c = -2$. Show the status of the stack after each step.

7. Use the stack algorithms in this chapter to convert the infix expression $a/b*c$ to postfix form. Be sure to account for left-to-right association. Show the status of the stack after each step.

8. Explain the significance of the precedence tests in the infix-to-postfix conversion algorithm. Why is a $\geq$ test used rather than a $>$ test?

9. Execute the HPAir algorithm with the map in Figure 7-17 for the following requests. Show the state of the stack after each step.

   a. Fly from *F* to *I*.

   b. Fly from *F* to *C*.

   c. Fly from *H* to *C*.

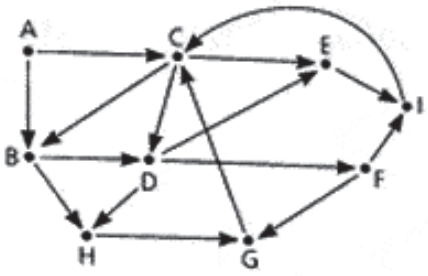


FIGURE 7-17

Flight map for Self-Test Exercise 9 and Exercise 15

Exercises

1. What makes a linked list a good choice for implementing a stack?

2. Compare the operations in the ADT stack with the ADT list. Which operations are basically the same? What operations differ? Explain your answer.

3. Suppose that you have a stack `aStack` and an empty auxiliary stack `auxStack`. Show how you can do each of the following tasks by using only the operations of the ADT stack:

   a. Display the contents of `aStack` in reverse order; that is, display the top last.

   b. Count the number of items in `aStack`, leaving `aStack` unchanged.

   c. Delete every occurrence of a specified item from `aStack`, leaving the order of the remaining items unchanged.

4. Recall the search method from the JCF class `Stack`:

```
public int search(Object o)
// Returns the 1-based position where an object is on this
// stack. The topmost item on the stack is considered to be at
// distance 1.
```

   a. Add this method to the `StackListBased` implementation given in this chapter.

   b. Add this method to the `StackArrayBased` implementation given in this chapter.

   c. Add this method to the `StackReferenceBased` implementation given in this chapter.

5. An operation that displays the contents of a stack can be useful during program debugging. Add a `display` method to the ADT stack such that

   a. The method is a client that only uses only ADT stack operations; that is, it is independent of the stack's implementation

   b. The method is within the ADT stack implementation and uses the reference-based implementation

6. Another operation that could be added to the ADT Stack is one that removes and discards the user specified number of elements from the top of the stack. Assume this operation is called `popAndDiscard` and that it does not return a value and accepts a parameter called count of data type `int`

   a. Add this operation to the `StackListBased` implementation given in this chapter.

   b. Add this operation to the `StackArrayBased` implementation given in this chapter.

   c. Add this operation to the `StackReferenceBased` implementation given in this chapter.

7. The diagram of a railroad switching system in Figure 7-18 is commonly used to illustrate the notion of a stack. Identify three stacks in the figure and show how they relate to one another. Suppose you had four train cars arrive in the order shown: A followed by B, then C, then D.

   a. How could you use the railroad switch to change the order so that A is still first, B is still second, but D is third, and C is fourth?

   b. How could you use the railroad switch to change the order of the cars so that B is first, D is second, A is third and C is fourth?

   c. Can any possible permutation of railroad cars be achieved with this switch? Justify your answer.

8. Suppose you have a stack in which the values 1 through 5 must be pushed on the stack in that order, but that an item on the stack can be popped and printed at any time. So for example, the operations

```
s.push(1)
s.push(2)
print s.pop()
```

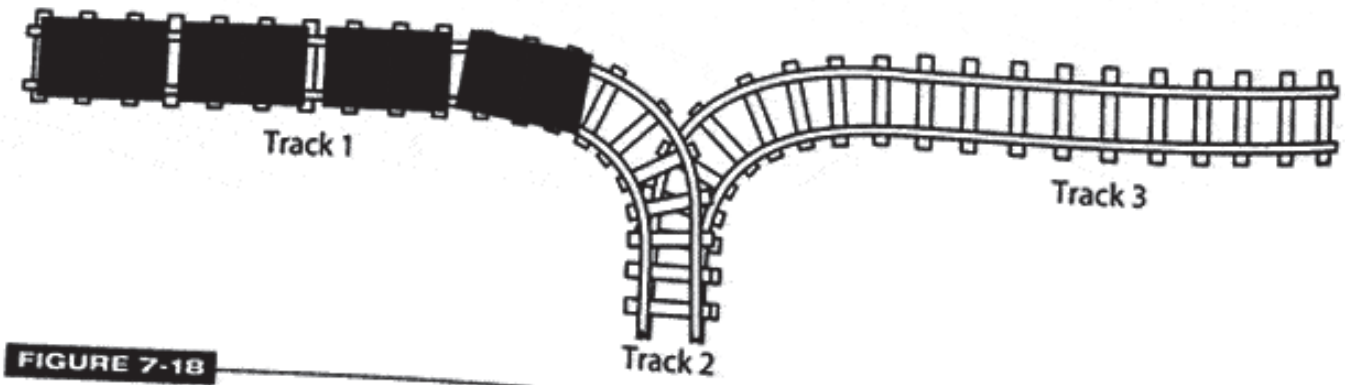


FIGURE 7-18
Railroad switching system for Exercise 4

```
s.push(3)
s.push(4)
print s.pop()
s.push(5)
print s.pop()
print s.pop()
print s.pop()
```

produce the sequence 2 4 5 3 1. Based on the constraints mentioned above, give the list of operations that would produce each of the following sequences. If it is not possible, state so.

a. 1 3 5 4 2

b. 2 3 4 5 1

c. 1 4 3 5 2

d. 5 4 2 3

e. Are there sequences that cannot occur? Explain why or why not.

9. What is the output of the following code for `n = 15`? `n = 65`? What does this program appear to do?

```
public static void mystery(int n) {
  StackInterface s = new StackReferenceBased();
  while (n > 0) {
    s.push(n % 8);
    n = n / 8;
  }
  while (!s.isEmpty())
    System.out.print(stack.pop());
  System.out.println();
} // end mystery
```

10. Consider the stack implementation that uses the ADT list to represent the items in the stack. Discuss the efficiency of the stack's insertion and deletion operations when the ADT list's implementation is

a. array based.

b. reference based.

11. The section "Developing an ADT During the Design of a Solution" described an algorithm ***readAndCorrect*** that reads a string of characters, correcting mistakes along the way.

a. For the following input line, trace the execution of ***readAndCorrect*** and show the contents of the stack at each step:

`ab←cde←←fgh←i`

b. The nature of the stack-based algorithm makes it simple to display the string in reverse order (as was done in ***displayBackward***), but somewhat harder to display it in its correct order. Write a pseudocode algorithm called ***displayForward*** that displays the string in its correct forward order.

c. Implement ***readAndCorrect*** and ***displayForward*** as Java methods.

12. Revise the pseudocode solution to the balanced-braces problem so that the expression can contain three types of delimiters: `()`, `[]`, and `{}`. Thus, `{ab(c[d])e}` is valid, but `{ab(c)}` is not.

13. For each of the following strings, trace the execution of the language-recognition algorithm described in the section "Recognizing Strings in a Language," and show the contents of the stack at each step.

    a. `xy$yxz`

    b. `xy$yx`

    c. `xyz$zxy`

    d. `zzz$zz`

    e. `xyzzy`

14. Write a pseudocode method that uses a stack to determine whether a string is in the language L, where

    a. $L = \{w : w$ contains equal numbers of A's and B's in any order$\}$

    b. $L = \{w : w$ is of the form $A^{2n}B^n$ for some $n \geq 0\}$

15. Write a method that uses a stack to determine whether a string is in the language L, where:

    $L = \{ww' : w$ is a string of characters
    $w' = reverse\ (w)\}$

    Note: The empty string, a string with less than 2 characters, or a string with an odd number of characters will not be in the language.

16. Evaluate the following postfix expressions by using the algorithm given in this chapter. Show the status of the stack after each step of the algorithm. Assume that division is integer division as in Java, and the identifiers have the following values: $a = 7$; $b = 3$; $c = 12$; $d = -5$; $e = 1$.

    a. *ab*cd+-*

    b. *abcd+-**

    c. *ab/c+d*e-*

17. Convert the following infix expressions to postfix form by using the algorithm given in this chapter. Show the status of the stack after each step of the algorithm.

    a. *a*b+c*

    b. *a/b*c*

    c. *a*(b+c)*

    d. *a+(b-c)*

    e. *a*(b+c-d)*

    f. *a/(b+c)/(d*e)*

g. $a-(b+c)/d+e$

h. $((a+b*(c-d))*e)-f$

18. Execute the HPAir algorithm with the map in Figure 7-17 (see Self-Test Exercise 9) for the following requests. Assume that the algorithm always flies to the alphabetically earliest unvisited city from the city on the top of the stack. Show the state of the stack after each step and indicate whether the flight is possible or not possible.

a. Fly from *A* to *I*.

b. Fly from *G* to *A*.

c. Fly from *H* to *I*.

d. Fly from *F* to *I*.

e. Fly from *I* to *G*.

19. As Chapter 4 pointed out, you can define ADT operations in a mathematically formal way by using axioms. For example, the following axioms formally define the ADT stack, where `stack` is an arbitrary stack and `item` is an arbitrary stack item.

```
(aStack.createStack()).isEmpty() = true
(aStack.push(item)).isEmpty() = false
(aStack.createStack()).pop() = error
(aStack.push(item)).pop() = aStack
(aStack.createStack()).peek() = error
(aStack.push(item)).peek() = item
```

You can use these axioms, for example, to prove that the stack defined by the sequence of operations

```
Create an empty stack
Push a 5
Push a 7
Push a 3
Pop (the 3)
Push a 9
Push a 4
Pop (the 4)
```

which you can write as

```
((((((aStack.createStack()).push(5)).push(7)).push(3)).
  pop()).push(9)).push(4)).pop()
```

is exactly the same as the stack defined by the sequence

```
Create an empty stack
Push a 5
Push a 7
Push a 9
```

which you can write as

```
(((aStack.createStack()).
```

Similarly, you can use the axioms to show that

```
((((((aStack.createStack()).push(1)).push(2)).pop()).
  push(3)).pop()).pop()).isEmpty()
```

is true.

a. The following representation of a stack as a sequence of *push* operations without any *pop* operations is called a **canonical form:**

```
(...(aStack.createStack()).push()).push()... ).push()
```

Prove that any stack is equal to a stack that is in canonical form.

b. Prove that the canonical form is unique. That is, a stack is equal to exactly one stack that is in canonical form.

c. Use the axioms to show formally that

```
((((((((((aStack.createStack()).push(6)).push(9)).
  pop()).pop()).push(2)).pop()).push(3)).push(1)).
  pop()).peek()
```

equals 3.

Programming Problems

1. The array-based implementation of the ADT stack in this chapter assumed a maximum stack size of 50 items. Modify this implementation so that each time the stack becomes full, the size of the array is doubled.

2. Implement the solution to the expanded balanced-braces problem in Exercise 9.

3. The section "Recognizing Strings in a Language" describes a recognition algorithm for the language

   $L = \{w\$w' : w \text{ is a possibly empty string of characters other than \$},$
   $w' = \text{reverse}(w)\}$

   Implement this algorithm.

4. Design and implement a class of postfix calculators. Use the algorithm given in this chapter to evaluate postfix expressions, as entered into the calculator. Use only the operators +, −, *, %, and /. Assume that the postfix expressions have single digit numbers in the expression and are syntactically correct.

5. The postfix calculator in Programming Problem 4 assumed single digit operands. Modify the calculator so that operators and operands may be separated by any number of spaces and

   a. the operands may be multi-digit integers.

   b. the operands may be multi-digit numbers with a decimal point.

6. Create a simple infix expressions that consist of sing

Design and implement a class for an infix calculator. Use the algorithms given in this chapter to convert the infix expression to postfix form and to evaluate the resulting postfix expression. Note that if the methods `evaluate` and `getPostfix` are called before the `convertPostfix` method, then the exception `IllegalStateException` should be thrown by these methods.

```
class Calculator {
  public Calculator(String exp)   // initializes infix expression
  public String toString()          // returns infix expression
  private boolean convertPostfix()  // creates postfix expression
                                    // returns true if successful

  // The following methods should throw IllegalStateException if
  // they are called before convertPostfix

  // returns the resulting postfix expression
  public String getPostfix() throws IllegalStateException

  // evaluates the expression
  public int evaluate() throws IllegalStateException
} //end Calculator
```

7. The infix-to-postfix conversion algorithm described in this chapter assumes single digit operands and that the given infix expression is syntactically correct. Repeat Programming Problem 6 with the following enhancements. If the expression has one of the errors mentioned, print out an appropriate error message, and where possible, indicate where the error occurred in the expression. If the expression is syntactically correct, evaluate the expression.

   a. Allow for any type of spacing between operands, operators, and parentheses.

   b. Allow for multi-digit integer operands. Even better, allow for multi-digit operands with a decimal point.

   c. The algorithms in the text assume that the given infix expression is syntactically correct. Watch for errors in the infix expression. Here are some examples:

      i. `a + 4` (Illegal character `a`)

      ii. `4 + 5 3` (Space between 5 and 3, a missing operator)

      iii. `4 + * 5 - 2` (Missing operand)

      iv. `) 2+3 (` (Improperly nested parenthesis)

      v. `( 2 + 3 ) * 5 )` (Mismatched parentheses-right or left)

      If an error is detected during the method `convertPostfix`, it should return false, but first print a message that identifies the error and, when possible, indicate where the error occurred in the expression. If the expression is not successfully converted, a call to `evaluate` or `getPostfix` should throw `IllegalStateException`.

8. Repeat Programming Problem 5, but use the following algorithm to evaluate an infix expression `infixExp`. The algorithm uses two stacks: One stack, `opStack`, contains operators, and the other stack, `valStack`, contains values of operands and intermediate results. Note that the algorithm treats parentheses as operators with the lowest precedence.

```
for (each character ch in infixExp) {
  switch (ch) {
    case ch is an operand, that is, a digit
      valStack.push(ch)
      break
    case ch is '('
      opStack.push(ch)
      break
    case ch is an operator
      if (opStack.isEmpty()) {
        opStack.push(ch)
      }
      else if (precedence(ch) >
                  precedence(top of opStack)) {
        opStack.push(ch)
      }
      else {
        while (!opStack.isEmpty() and
          precedence(ch) <= precedence(top of opStack)) {
          Execute
        }  // end while
        opStack.push(ch)
      }  // end if
      break

    case ch is ')'
      while (top of opStack is not '(') {
        Execute
      }  // end while
      opStack.pop()
      break
  }  // end switch
}  // end for

while (!opStack.isEmpty()) {
  Execute
}  // end while
result = valStack.peek()
```

Note that `Execute` means

```
operand2 = valStack.pop()
operand1 = valStack.pop()
op = opStack.pop()
result = operand1 op operand2
valStack.push(result)
```

9. In the chapter, we examined one strategy to evaluate an infix expression—first convert the infix expression to a postfix expression, then evaluate the resulting postfix expression. An alternative strategy would be to convert the infix expression to prefix, and then evaluate the resulting prefix expression.

   The following pseudocode algorithm for converting an infix expression to prefix is similar to, but not quite the same as, the infix to postfix conversion algorithm:

```
Initialize the string temp to the null string
Reverse the characters in the infix expression
```

```
for (each character ch in the reversed infix expression) {
  switch(ch) {
    case ch is an operand:
      Append ch to the end of temp expression
    case ch is ')':
      Push ch on the stack
    case ch is '(':
       Pop stack and append item to output expression until the
         matching ')' is popped off the stack
    case ch is an operator:
      Pop operators off stack and append to temp expression as
        appropriate (similar to postfix conversion)
      Push ch on the stack
  } // end switch
  end for
append remaining stack contents to output expression
while (stack is not empty) {
  Pop stack and append to temp expression
  end while
reverse temp expression to produce prefix expression
```

Also note that evaluating a prefix expression is almost the same as evaluating a postfix expression, with one small change—with prefix expressions, you start at the end of the expression. For example:

```
Postfix : 3 5 +
```

You start at the beginning of the expression, moving forward through the expression, pushing operands, and popping the operands when the operators appear, then pushing the result.

```
Prefix: + 3 5
```

You start at the end of the expression, moving backward through the expression, pushing operands, and popping the operands when the operators appear, then pushing the result.

Design and implement a class (as shown next) for an infix calculator based on prefix expressions. Use the algorithms given above to convert the infix expression to prefix form and to evaluate the resulting prefix expression. Note that if the methods ***evaluate*** and ***getPrefix*** are called before the ***convertPrefix*** method, then the exception ***IllegalStateException*** should be thrown by these methods.

```
class Calculator {
  public Calculator(String exp)   // initializes infix expression
  public String toString()          // returns infix expression
  private boolean convertPrefix()   // creates prefix expression
                                    // returns true if successful

  // The following methods should throw IllegalStateException if
  // they are called before convertPrefix

  // returns the resulting prefix expression
  public String getPrefix() throws IllegalStateException
```

```
    // evaluates the expression
    public int evaluate() throws IllegalStateException
} //end Calculator
```

10. Using stacks, write a nonrecursive version of the method *solveTowers*, as defined in Chapter 3.

11. Complete the solution to the HPAir problem. The input to the program consists of three text files, as follows:

cityFile	Each line contains the name of a city that HPAir serves. The names are in alphabetical order.
flightFile	Each line contains a pair of city names that represents the origin and destination of one of HPAir's flights.
requestFile	Each line contains a pair of city names that represents a request to fly from some origin to some destination.

You can make the following assumptions:

- Each city name contains at most 15 characters. Pairs of city names are separated by a comma.
- HPAir serves at most 20 cities.
- The input data is correct.

For example, the input files could appear as

```
cityFile:     Albuquerque
              Chicago
              San Diego

flightFile:   Chicago,       San Diego
              Chicago,       Albuquerque
              Albuquerque,   Chicago

requestFile:  Albuquerque,   San Diego
              Albuquerque,   Paris
              San Diego,     Chicago
```

For this input, the program should produce the following output:

```
Request is to fly from Albuquerque to San Diego.
HPAir flies from Albuquerque to San Diego.

Request is to fly from Albuquerque to Paris.
Sorry. HPAir does not serve Paris.

Request is to fly from San Diego to Chicago.
Sorry. HPAir does not fly from San Diego to Chicago.
```

Begin by implementing the ADT flight map as the Java class *Map*. Use the nonrecursive version of *isPath*. Since *getNextCity* is the primary operation that the search algorithm performs on the flight map, you should choose an implementation that will efficiently determine which cities are adjacent to a given city. If there are *N*

cities, you can use N linked lists to represent the flight map. You place a node on list i for city j if and only if there is a directed path from city i to city j. Such a data structure is called an **adjacency list;** Figure 7-19 illustrates an adjacency list for the flight map in Figure 7-10. Chapter 14 discusses adjacency lists further when it presents ways to represent graphs. At that time, you will learn why an adjacency list is a good choice for the present program.

Although you can implement the adjacency list from scratch, you should also consider using N instances of `ListReferenceBased`, which has a reference-based implementation.

You must also create a class `City` that implements the `java.lang.Comparable` interface to store the city name. The class `City` and the previously described adjacency list are the underlying data structures for the ADT flight map.

To simplify reading the input text files, define a class that includes the following methods:

```
-getName():String
   Gets a name from the next line in a text file.

-getNamePair():String
   Returns a string containing the two names from the next
   line in a text file.
```

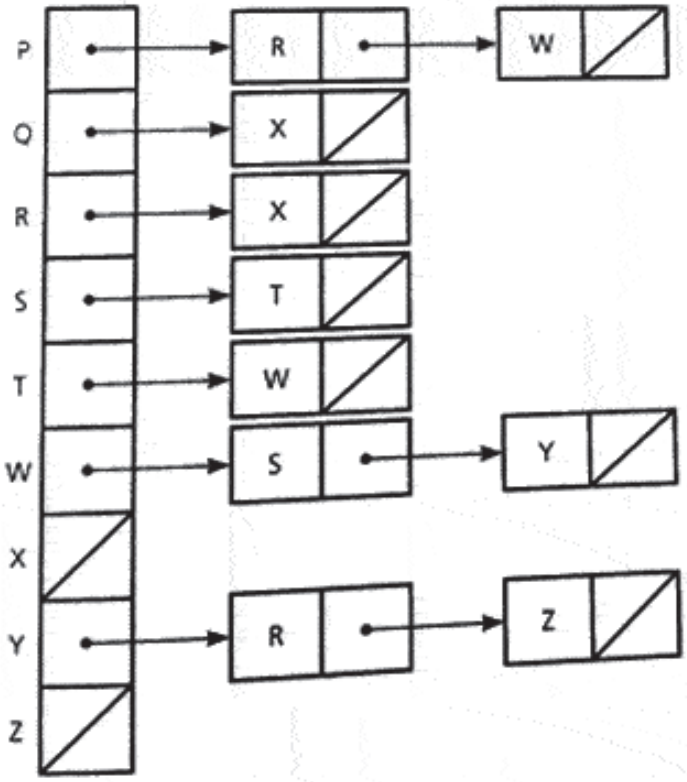


FIGURE 7-19

Adjacency list for the flight map in Figure 7-10

12. In the implementation of the HPAir problem (see Programming Problem 10), the search for the next unvisited city adjacent to a city i always starts at the beginning of the i^{th} linked list in the adjacency list. This approach is actually a bit inefficient, because once the search visits a city, the city can never become unvisited. Modify the program so that the search for the next city begins where the last search left off. That is, maintain an array of `tryNext` references into the adjacency list.

13. Implement an expanded version of the HPAir problem. In addition to the "from" and "to" cities, each line of input contains a flight number (an integer) and the cost of the flight (an integer). Modify the HPAir program so that it will produce a complete itinerary for each request, including the flight number of each flight, the cost of each flight, and the total cost of the trip.

    For example, the input files could appear as

```
cityFile:       Albuquerque
                Chicago
                San Diego

flightFile:     Chicago,        San Diego    703   325
                Chicago,        Albuquerque  111   250
                Albuquerque,    Chicago      178   250

requestFile:    Albuquerque,    San Diego
                Albuquerque,    Paris
                San Diego,      Chicago
```

    For this input, the program should produce the following output:

```
Request is to fly from Albuquerque to San Diego.
Flight #178 from Albuquerque to Chicago   Cost: $250
Flight #703 from Chicago to San Diego     Cost: $325
Total Cost ..............  $575
Request is to fly from Albuquerque to Paris.
Sorry. HPAir does not serve Paris.
Request is to fly from San Diego to Chicago.
Sorry. HPAir does not fly from San Diego to Chicago.
```

When the nonrecursive `isPath` method finds a sequence of flights from the origin city to the destination city, its stack contains the corresponding path of cities. The stumbling block to reporting this path is that the cities appear in the stack in reverse order; that is, the destination city is at the top of the stack and the origin city is at the bottom. For example, if you use the program to find a path from city P to city Z in Figure 7-10, the final contents of the stack will be P-W-Y-Z, with Z on top. You want to display the origin city P first, but it is at the bottom of the stack. If you restrict yourself to the stack operations, the only way that you can write the path in its correct order is first to reverse the stack by popping it onto a temporary stack and then to write the cities as you pop them off the temporary stack. Note that this approach requires that you process each city on the path twice.

Evidently a stack is not the appropriate ADT for the problem of writing the path of cities in the correct order; the appropriate ADT is a **traversable stack**. In addition to the standard stack operations, `isEmpty`, `push`, `pop`, and `peek`, a traversable

stack includes the operation **`traverse`**. This operation begins at one end of the stack and *visits* each item in the stack until it reaches the other end of the stack. For this project, you want **`traverse`** to begin at the bottom of the stack and move toward the top.

14. What modifications to Programming Problem 13 are required to find a least-cost trip for each request? How can you incorporate time considerations into the problem?

CHAPTER 8

Queues

Whereas a stack's behavior is characterized as last-in, first-out, a queue's behavior is characterized as first-in, first-out. This chapter defines the queue's operations and discusses strategies for implementing them. As you will see, queues are common in everyday life. Their first-in, first-out behavior makes them appropriate ADTs for situations that involve waiting. Queues are also important in simulation, a technique for analyzing the behavior of complex systems. This chapter uses a queue to model the behavior of people in a line.

8.1 The Abstract Data Type Queue

A **queue** is like a line of people. The first person to join a line is the first person served and is thus the first to leave the line. New items enter a queue at its **back,** or **rear,** and items leave a queue from its **front.** Operations on a queue occur only at its two ends. This characteristic gives a queue its **first-in, first-out (FIFO)** behavior. In contrast, you can think of a stack as having only one end, because all operations are performed at the top of the stack. This characteristic gives a stack its last-in, first-out behavior.

FIFO. The first item inserted into a queue is the first item out

As an abstract data type, the queue has the following operations:

KEY CONCEPTS

ADT Queue Operations

1. Create an empty queue.
2. Determine whether a queue is empty.
3. Add a new item to the queue.
4. Remove from the queue the item that was added earliest.
5. Remove all the items from the queue.
6. Retrieve from the queue the item that was added earliest.

Queues occur in everyday life

Queues are appropriate for many real-world situations. You wait in a queue—that is, a line—to buy a movie ticket, to check out at the book store, or to use an automatic teller machine. The person at the front of the queue is served, while new people join the queue at its back. Even when you call an airline to make a reservation, your call actually enters a queue while you wait for the next available agent.

Queues have applications in computer science

Queues also have applications in computer science. When you print an essay, the computer sends lines faster than the printer can print them. The lines are held in a queue for the printer, which removes them in FIFO order. If you share the printer with other computers, your request to print enters a queue to wait its turn.

Since all of these applications involve waiting, people study them to see how to reduce the wait. Such studies are called **simulations,** and they typically use queues. Later, this chapter examines a simulation of a line of customers at a bank.

The following pseudocode describes the operations for the ADT queue in more detail, and Figure 8-1 shows a UML diagram for the class *Queue*. As we did for the ADT stack, we specify that the remove operation both retrieves and then removes the item at the front of the queue.

Figure 8-2 illustrates these operations with a queue of integers. Notice that *enqueue* inserts an item at the back of the queue and that *peek* looks at the item at the front of the queue, whereas *dequeue* deletes the item at the front of the queue.

KEY CONCEPTS

Pseudocode for the ADT Queue Operations

```
//QueueItemType is the type of the items stored in the queue
+createQueue()
// Creates an empty queue.
+isEmpty():boolean {query}
// Determines whether a queue is empty.
+enqueue(in newItem:QueueItemType) throws QueueException
// Adds NewItem at the back of a queue. Throws
// QueueException if the operation is not successful.
+dequeue():QueueItemType throws QueueException
// Retrieves and removes the front of a queue—the
// item that was added earliest. Throws QueueException
// if the operation is not successful.
+dequeueAll()
// Removes all items from a queue
+peek():QueueItemType {query} throws QueueException
// Retrieves the front of a queue. That is,
// retrieves the item that was added earliest.
// Throws QueueException if the retrieval is not
// successful. The queue is unchanged.
```

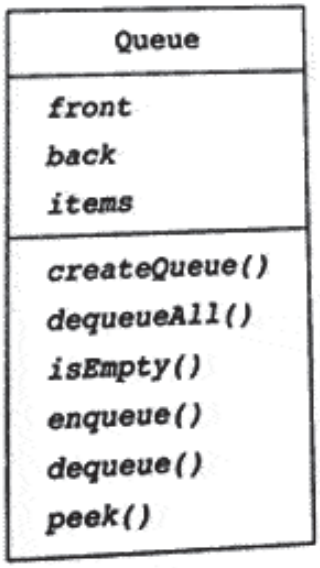


FIGURE 8-1

UML diagram for the class ***Queue***

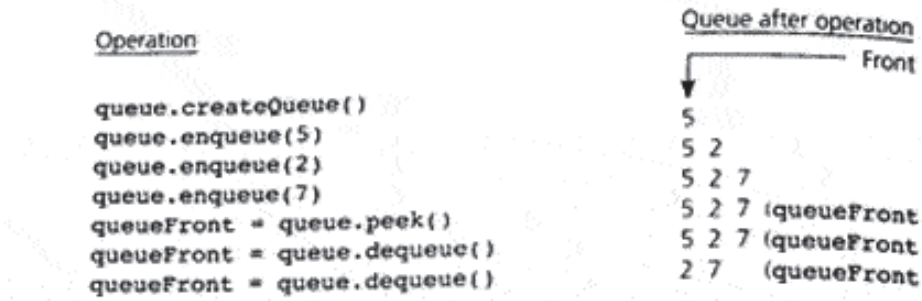


FIGURE 8-2

Some queue operations

8.2 Simple Applications of the ADT Queue

This section presents two simple applications of the ADT queue. The applications use the operations of the ADT queue independently of their implementations.

Reading a String of Characters

When you enter characters at a keyboard, the system must retain them in the order in which you typed them. It could use a queue for this purpose, as the following pseudocode indicates:

A queue can retain characters in the order in which you type them

```
// read a string of characters from a
// single line of input into a queue
aQueue.createQueue()
while (not end of line) {
   Read a new character ch
   aQueue.enqueue(ch)
}  // end while
```

Once the characters are in a queue, the system can process them as necessary. For example, if you had typed an integer—without any mistakes, but possibly preceded or followed by blanks—the queue would contain digits and possibly blanks. If the digits are 2, 4, and 7, the system could convert them into the decimal value 247 by computing

$$10 * (10 * 2 + 4) + 7$$

The following pseudocode performs this conversion in general:

```
// convert digits in queue aQueue into a decimal integer n

// get first digit, ignoring any leading blanks
do  {
  ch = aQueue.dequeue()
}  while (ch is blank)
```

```
// Assertion: ch contains first digit
// compute n from digits in queue
n = 0
done = false
do {
  n = 10 * n + integer that ch represents
  if (!aQueue.isEmpty()) {
    ch = aQueue.dequeue()
  }
  else {
    done = true
  } // end if
} while (!done and ch is a digit)
// Assertion: n is result
```

Recognizing Palindromes

You can use a queue in conjunction with a stack to recognize palindromes

Recall from Chapter 6 that a palindrome is a string of characters that reads the same from left to right as it does from right to left. In the previous chapter, you learned that you can use a stack to reverse the order of occurrences. You should realize by now that you can use a queue to preserve the order of occurrences. Thus, you can use both a queue and a stack to determine whether a string is a palindrome.

As you traverse the character string from left to right, you can insert each character into both a queue and a stack. Figure 8-3 illustrates the result of this action for the string abcbd, which is not a palindrome. You can see that the first character in the string is at the front of the queue and the last character in the string is at the top of the stack. Thus, characters removed from the queue

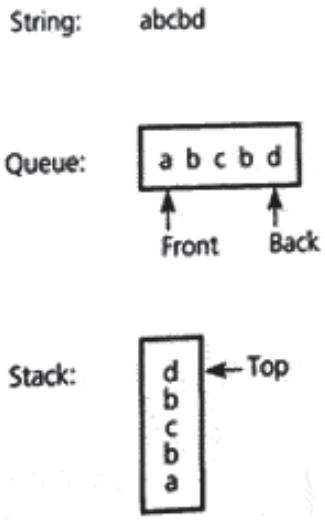


FIGURE 8-3

The results of inserting a string into both a queue and a stack

will occur in the order in which they appear in the string; characters removed from the stack will occur in the opposite order.

Knowing this, you can compare the characters at the front of the queue and the top of the stack. If the characters are the same, you can delete them. You repeat this process until either the ADTs become empty, in which case the original string is a palindrome, or the two characters are not the same, in which case the string is not a palindrome.

The following is a pseudocode version of a nonrecursive recognition algorithm for the language of palindromes:

A nonrecursive recognition algorithm for palindromes

```
+isPal(in str:String):boolean
// Determines whether str is a palindrome.

  // create an empty queue and an empty stack
  aQueue.createQueue()
  aStack.createStack()

  // insert each character of the string into both
  // the queue and the stack
  length = the length of str
  for (i = 1 through length) {
     nextChar = ith character of str
     aQueue.enqueue(nextChar)
     aStack.push(nextChar)
  }  // end for

  // compare the queue characters with the stack
  // characters
  charactersAreEqual = true
  while (aQueue is not empty and charactersAreEqual is true) {
     queueFront = aQueue.dequeue()
     stackTop = aStack.pop()
     if (queueFront not equal to stackTop) {
        charactersAreEqual = false
     }  // end if
  }  // end while
  return charactersAreEqual
```

8.3 Implementations of the ADT Queue

This section develops three Java implementations of the ADT queue. The first uses a linked list to represent the queue, the second uses an array, and the third uses the ADT list. The following interface `QueueInterface` is used to provide a common specification for the three implementations. Note that *enqueue*, *dequeue*, and *peek* may throw `QueueException`.

The `QueueException` class, which is similar to the `StackException` class developed in Chapter 7, appears next.

```
public interface QueueInterface {

  public boolean isEmpty();
  // Determines whether a queue is empty.
  // Precondition: None.
  // Postcondition: Returns true if the queue is empty;
  // otherwise returns false.

  public void enqueue(Object newItem) throws QueueException;
  // Adds an item at the back of a queue.
  // Precondition: newItem is the item to be inserted.
  // Postcondition: If the operation was successful, newItem
  // is at the back of the queue. Some implementations may
  // throw QueueException if newItem cannot be added to the
  // queue.

  public Object dequeue() throws QueueException;
  // Retrieves and removes the front of a queue.
  // Precondition: None.
  // Postcondition: If the queue is not empty, the item
  // that was added to the queue earliest is returned and
  // the item is removed. If the queue is empty, the
  // operation is impossible and QueueException is thrown.

  public void dequeueAll();
  // Removes all items of a queue.
  // Precondition: None.
  // Postcondition: The queue is empty.

  public Object peek() throws QueueException;
  // Retrieves the item at the front of a queue.
  // Precondition: None.
  // Postcondition: If the queue is not empty, the item
  // that was added to the queue earliest is returned.
  // If the queue is empty, the operation is impossible
  // and QueueException is thrown.
} // end QueueInterface
```

The *QueueException* class, which is similar to the *StackException* class developed in Chapter 7, appears next.

```
public class QueueException extends RuntimeException {

  public QueueException(String s) {
    super(s);
  } // end constructor
} // end QueueException
```

For queues, the reference-based implementation is a bit more straightforward than the array-based one, so we start with it.

A Reference-Based Implementation

A reference-based implementation of a queue could use a linear linked list with two external references, one to the front and one to the back, as Figure 8-4a illustrates.[1] However, as Figure 8-4b shows, you can actually get by with a single external reference—to the back—if you make the linked list circular.

A circular linked list can represent a queue

When a circular linked list represents a queue, the node at the back of the queue references the node at the front. Thus,

`lastNode` references the node at the back of the queue, and

`lastNode.next` references the node at the front

Insertion at the back and deletion from the front are straightforward. Figure 8-5 illustrates the addition of an item to a nonempty queue. Inserting the new node, which `newNode` references, at the back of the queue requires three reference changes: the next reference in the new node, the next reference in the back node, and the external reference `lastNode`. Figure 8-5 depicts

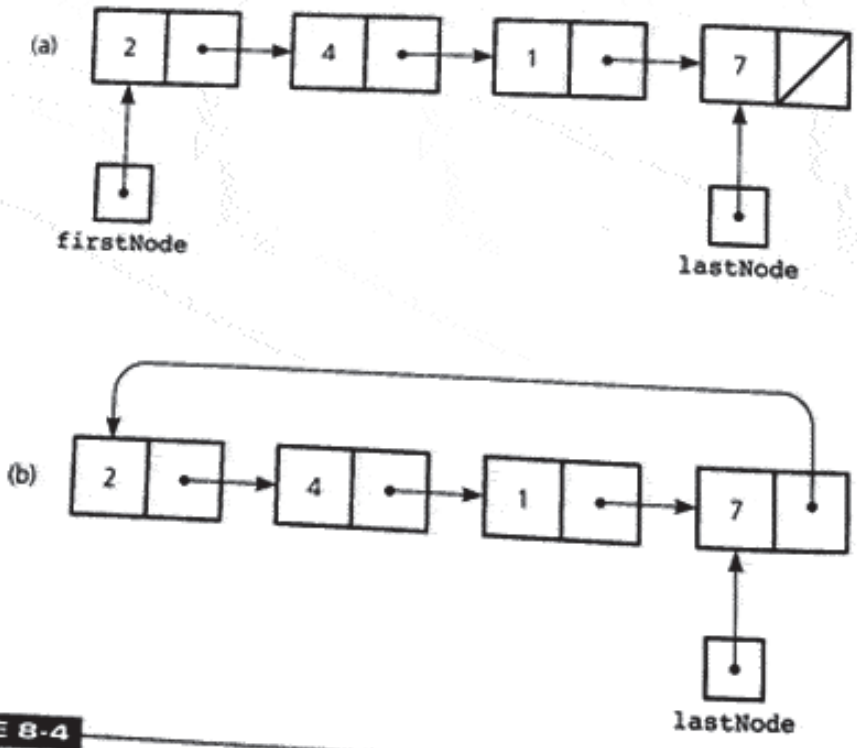


FIGURE 8-4

A reference-based implementation of a queue: (a) a linear linked list with two external references; (b) a circular linear linked list with one external reference

1. Programming Problem 1 asks you consider the details of this implementation.

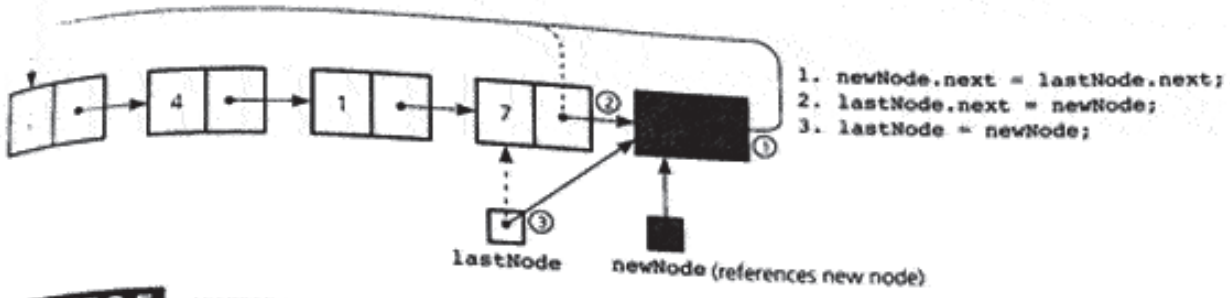


FIGURE 8-5
Inserting an item into a nonempty queue

these changes and indicates the order in which they must occur. (The dashed lines indicate reference values before the changes.) The addition of an item to an empty queue is a special case, as Figure 8-6 illustrates.

Deletion from the front of the queue is simpler than insertion at the back. Figure 8-7 illustrates the removal of the front item of a queue that contains more than one item. Notice that you need to change only one reference within the queue. Deletion from a queue of one item is a special case that sets the external reference `lastNode` to `null`.

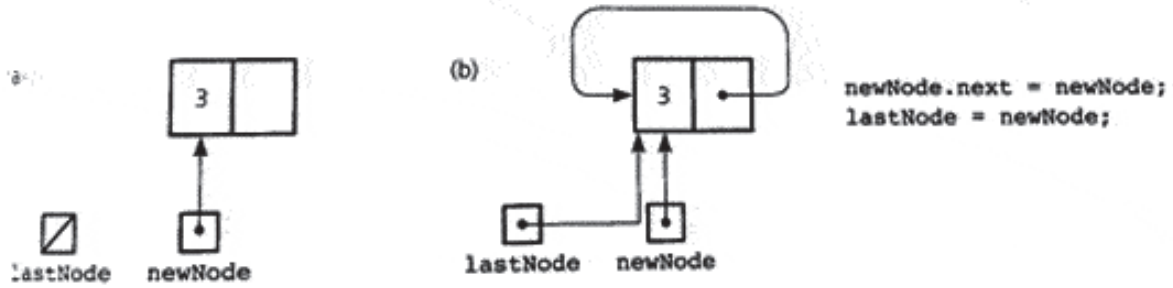


FIGURE 8-6
Inserting an item into an empty queue: (a) before insertion; (b) after insertion

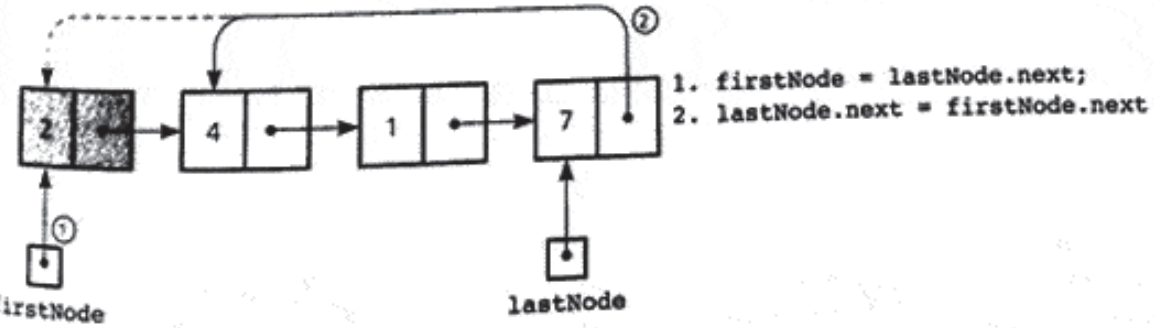


FIGURE 8-7
Deleting an item from a queue of more than one item

The following class is a reference-based implementation of the ADT queue. The implementation uses the *Node* class developed in Chapter 5.

```
public class QueueReferenceBased implements QueueInterface {
  private Node lastNode;

  public QueueReferenceBased() {
    lastNode = null;
  }  // end default constructor

  // queue operations:
  public boolean isEmpty() {
    return lastNode == null;
  }  // end isEmpty

  public void dequeueAll() {
    lastNode = null;
  }  // end dequeueAll

  public void enqueue(Object newItem) {
    Node newNode = new Node(newItem);

    // insert the new node
    if (isEmpty()) {
      // insertion into empty queue
      newNode.next = newNode;
    }
    else {
      // insertion into nonempty queue
      newNode.next = lastNode.next;
      lastNode.next = newNode;
    }  // end if

    lastNode = newNode;  // new node is at back
  }  // end enqueue

  public Object dequeue() throws QueueException {
    if (!isEmpty()) {
      // queue is not empty; remove front
      Node firstNode = lastNode.next;
      if (firstNode == lastNode) { // special case?
        lastNode = null;                // yes, one node in queue
      }
      else {
        lastNode.next = firstNode.next;
      }  // end if
```

```
      return firstNode.item;
    }
    else {
      throw new QueueException("QueueException on dequeue:"
                               + "queue empty");
    } // end if
  } // end dequeue

  public Object peek() throws QueueException {
    if (!isEmpty()) {
      // queue is not empty; retrieve front
      Node firstNode = lastNode.next;
      return firstNode.item;
    }
    else {
      throw new QueueException("QueueException on peek:"
                               + "queue empty");
    } // end if
  } // end peek

} // end QueueReferenceBased
```

A program that uses this implementation could begin as follows:

```
public class QueueTest {
  public static void main(String[] args) {
    QueueReferenceBased aQueue =
                                  new QueueReferenceBased();
    for (int i = 0; i < 9; i++) {
      aQueue.enqueue(new Integer(i));
    } // end for
    . . .
  } // end main
} // end QueueTest
```

An Array-Based Implementation

For applications in which a fixed-sized queue does not present a problem, you can use an array to represent a queue. As Figure 8-8a illustrates, a naive array-based implementation of a queue might include the following definitions:

A naive array-based implementation of a queue

```
final int MAX_QUEUE = maximum-size-of-queue;

Object[] items;
int      front;
int      back;
```

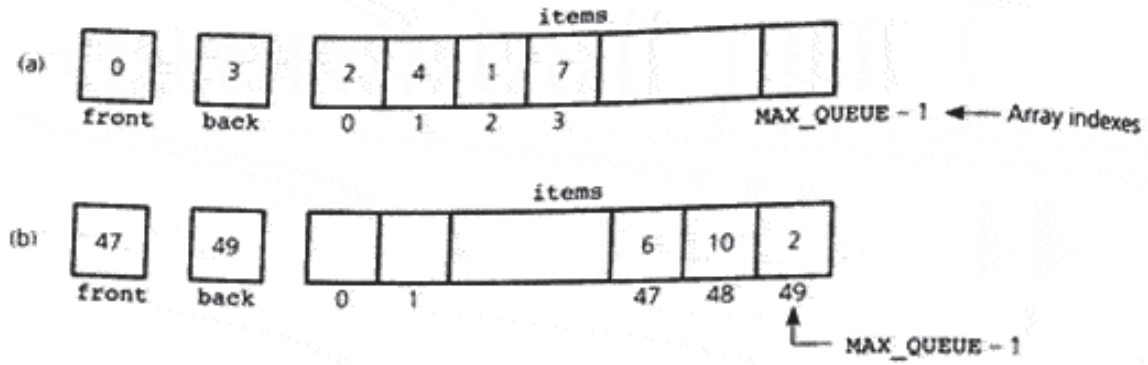


FIGURE 8-8

(a) A naive array-based implementation of a queue; (b) rightward drift can cause the queue to appear full

The indexes of the front and back items in the queue are, respectively, `front` and `back`. Initially, `front` is 0 and `back` is -1. To insert a new item into the queue, you increment `back` and place the item in `items[back]`. To delete an item, you simply increment `front`. The queue is empty whenever `back` is less than `front`. The queue is full when `back` equals `MAX_QUEUE - 1`.

Rightward drift can cause a queue-full condition even though the queue contains few entries

The problem with this strategy is **rightward drift**—that is, after a sequence of additions and removals, the items in the queue will drift toward the end of the array, and `back` could equal `MAX_QUEUE - 1` even when the queue contains only a few items. Figure 8-8b illustrates this situation.

Shifting elements to compensate for rightward drift is

One possible solution to this problem is to shift array elements to the left, either after each deletion or whenever `back` equals `MAX_QUEUE - 1`. This solution guarantees that the queue can always contain up to `MAX_QUEUE` items. Shifting is not really satisfactory, however, as it would dominate the cost of the implementation.

A circular array eliminates rightward drift

A much more elegant solution is possible by viewing the array as circular, as Figure 8-9 illustrates. You advance the queue indexes `front` (to delete an item) and `back` (to insert an item) by moving them clockwise around the array. Figure 8-10 illustrates the effect of a sequence of three queue operations on

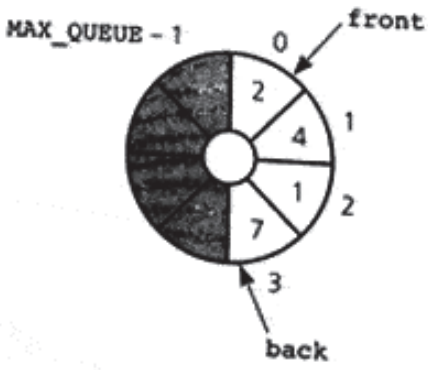


FIGURE 8-9

A circular implementation of a queue

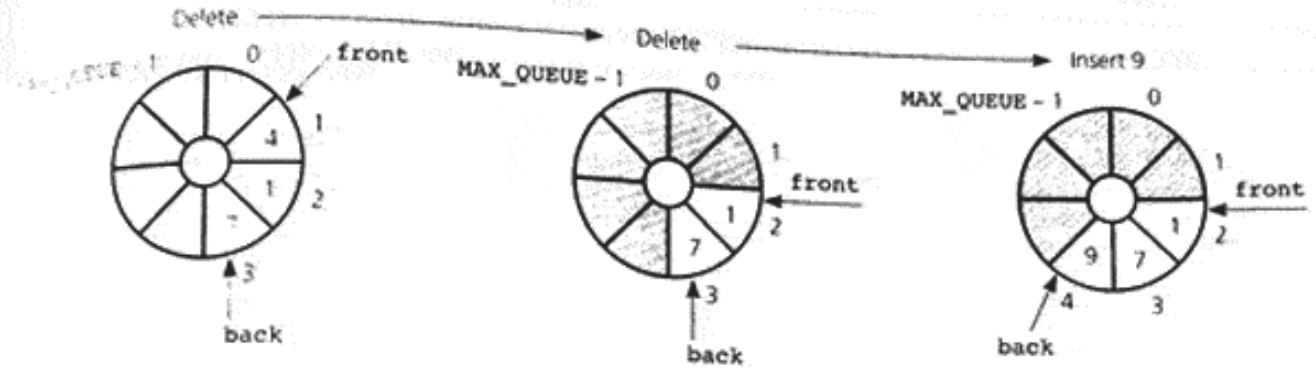


FIGURE 8-10

Effects of some operations on the queue in Figure 8-9

front, *back*, and the array. When either *front* or *back* advances past *MAX_QUEUE* – 1, it wraps around to 0. This wraparound eliminates the problem of rightward drift, which occurred in the previous implementation, because now the circular array has no end.

The only difficulty with this scheme involves detecting the queue-empty and queue-full conditions. It seems reasonable to select as the queue-empty condition

front is one slot ahead of *back*

since this appears to indicate that *front* "passes" *back* when the queue becomes empty, as Figure 8-11a depicts. However, it is also possible that this condition signals a full queue: Because the queue is circular, *back* might in fact "catch up" with *front* as the queue becomes full; Figure 8-11b illustrates this situation.

front and **back** cannot be used to distinguish between queue-full and queue-empty conditions

Obviously, you need a way to distinguish between the two situations. One such way is to keep a count of the number of items in the queue. Before inserting into the queue, you check to see if the count is equal to *MAX_QUEUE*; if it is, the queue is full. Before deleting an item from the queue, you check to see if the count is equal to zero; if it is, the queue is empty.

By counting queue items, you can detect queue-full and queue-empty conditions

To initialize the queue, you set *front* to 0, *back* to *MAX_QUEUE* – 1, and *count* to 0. You obtain the wraparound effect of a circular queue by using modulo arithmetic (that is, the Java % operator) when incrementing *front* and *back*. For example, you can insert *newItem* into the queue by using the statements

Initialize ***front***, ***back***, and ***count***

```
back = (back+1) % MAX_QUEUE;
items[back] = newItem;
++count;
```

Inserting into a queue

Notice that if *back* equaled *MAX_QUEUE* – 1 before the insertion of *newItem*, the first statement, *back = (back+1) % MAX_QUEUE*, would have the effect of wrapping *back* around to location 0.

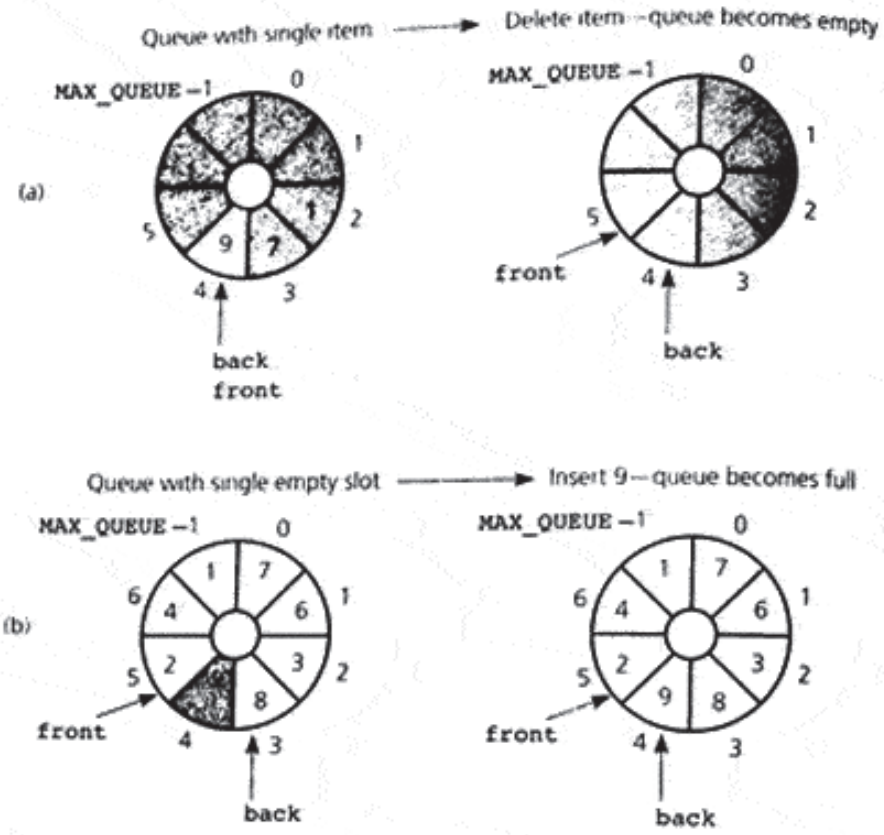


FIGURE 8-11

(a) ***front*** passes ***back*** when the queue becomes empty; (b) ***back*** catches up to ***front*** when the queue becomes full

Similarly, you can delete the item at the front of the queue by using the statements

Deleting from a queue

```
front = (front+1) % MAX_QUEUE;
--count;
```

The following Java class is an array-based implementation of the ADT queue that uses a circular array as just described. Preconditions and postconditions have been omitted to save space but are the same as those given in the *QueueInterface* specification.

```
public class QueueArrayBased implements QueueInterface {
  private final int MAX_QUEUE = 50; // maximum size of queue
  private Object[] items;
  private int front, back, count;

  public QueueArrayBased() {
    items = new Object[MAX_QUEUE];
```

```
    front = 0;
    back = MAX_QUEUE-1;
    count = 0;
  } // end default constructor

  // queue operations:
  public boolean isEmpty() {
    return count == 0;
  } // end isEmpty

  public boolean isFull() {
    return count == MAX_QUEUE;
  } // end isFull

  public void enqueue(Object newItem) throws QueueException {
    if (!isFull()) {
      back = (back+1) % (MAX_QUEUE);
      items[back] = newItem;
      ++count;
    }
    else {
      throw new QueueException("QueueException on enqueue: "
                               + "Queue full");
    } // end if
  } // end enqueue
  public Object dequeue() throws QueueException {
    if (!isEmpty()) {
      // queue is not empty; remove front
      Object queueFront = items[front];
      front = (front+1) % (MAX_QUEUE);
      --count;
      return queueFront;
    }
    else {
      throw new QueueException("QueueException on dequeue: "
                               + "Queue empty");
    }   // end if
  } // end dequeue
  public void dequeueAll() {
    items = new Object[MAX_QUEUE];
    front = 0;
    back = MAX_QUEUE-1;
    count = 0;
  } // end dequeueAll

  Public Object peek() throws QueueException {
    if (!isEmpty()) {
```

```
      // queue is not empty; retrieve front
      return items[front];
    }
    else {
      throw new QueueException("Queue exception on peek: " +
                               + "Queue empty");

    }  // end if
  }  // end peek

} // end QueueArrayBased
```

A **full** flag can replace the counter

Several commonly used variations of this implementation do not require a count of the number of items in the queue. One approach uses a flag *full* to distinguish between the full and empty conditions. The expense of maintaining a *full* flag is about the same as that of maintaining a counter, however. A faster implementation declares *MAX_QUEUE* + 1 locations for the array *items*, but uses only *MAX_QUEUE* of them for queue items. You sacrifice one array location and make *front* the index of the location before the front of the queue. As Figure 8-12 illustrates, the queue is full if

front equals *(back+1) % (MAX_QUEUE+1)*

but the queue is empty if

front equals *back*

Using an extra array location is more time-efficient

This implementation does not have the overhead of maintaining a counter or flag, and so is more efficient time-wise. For the standard data types, the implementation requires the same space as either the counter or the flag implementation (why?). Programming Problems 3 and 4 discuss these two alternate implementations further.

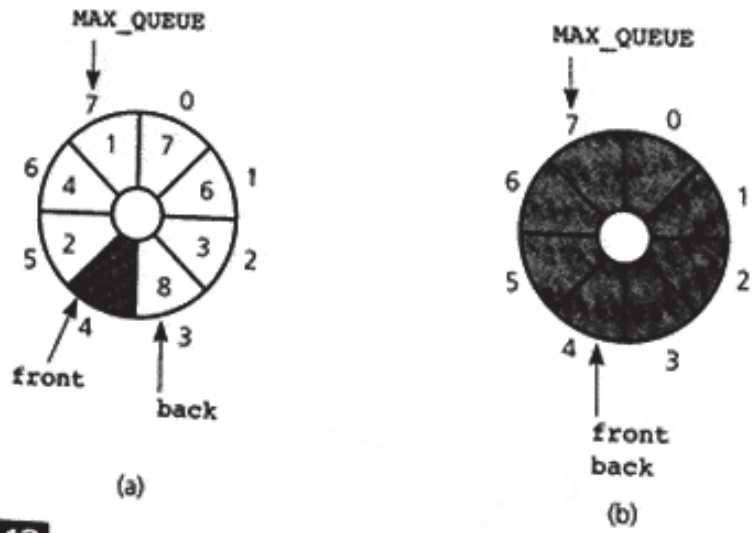


FIGURE 8-12

A more efficient circular implementation: (a) a full queue; (b) an empty queue

An Implementation That Uses the ADT List

You can use the ADT list to represent the items in a queue, as Figure 8-13 illustrates. If the item in position 1 of a list represents the front of the queue, you can implement the operation `dequeue()` as the list operation `remove(0)` and the operation `peek()` as `get(0)`. Similarly, if you let the item at the end of the list represent the back of the queue, you can implement the operation `enqueue(newItem)` as the list operation `add(size(), newItem)`.

Recall that Chapters 4 and 5 presented the ADT list as an implementation of the interface `ListInterface`. (See, for example, page 231.) The following class for the ADT queue uses an instance of the reference-based implementation `ListReferenceBased` to represent the queue. Preconditions and postconditions are omitted to save space but are the same as those given earlier in this chapter.

```
public class QueueListBased implements QueueInterface {
 private ListInterface aList;

 public QueueListBased() {
   aList = new ListReferenceBased();
 } // end default constructor

   queue operations:
 public boolean isEmpty() {
   return aList.isEmpty();
 } // end isEmpty

 public void enqueue(Object newItem) {
   aList.add(aList.size(), newItem);
 } // end enqueue

 public Object dequeue() throws QueueException {
   if (!isEmpty()) {
     // queue is not empty; remove front
     Object queueFront = aList.get(0);
     aList.remove(0);
     return queueFront;
   }
```

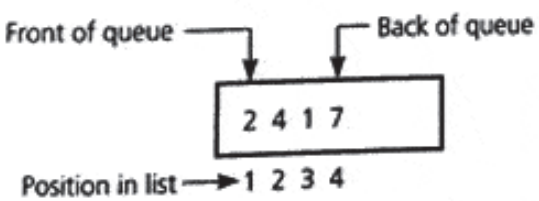


FIGURE 8-13

An implementation that uses the ADT list

```
      else {
        throw new QueueException("Queue exception on dequeue: "
                                 + " queue empty");
      } // end if
    } // end dequeue

    public void dequeueAll() {
      aList.removeAll();
    } // end dequeueAll

    public Object peek() throws QueueException {
      if (!isEmpty()) {
        // queue is not empty; retrieve front
        return aList.get(0);
      }
      else {
        throw new QueueException("Queue exception on peek "
                                 + "queue empty");
      } // end if
    } // end peek

} // end QueueListBased
```

As was true for the analogous implementation of the ADT stack in Chapter 7, implementing the queue is simple once you have implemented the ADT list. Exercise 6 at the end of this chapter asks you to consider the efficiency of this implementation.

The JCF Interfaces *Queue* and *Deque*

The Java Collections Framework (JCF) contains two interfaces for queues: *Queue* and *Deque* (usually pronounced "deck"). Like the *List* interface, *Queue* is derived from the interface *Collection*, and thus inherits all of the methods defined in *Collection*. The *Deque* interface is derived from the *Queue* interface and is for collections that support element insertion and removal at both ends. The name *Deque* is short for "double ended queue."

The *Queue* Interface. The *Queue* interface extends *Collection* with the following methods:

```
public interface Queue<E> extends Collection<E> {

  E element() throws NoSuchElementException;
    // Retrieves, but does not remove, the head of this queue.
    // If this queue is empty, throws NoSuchElementException.

  boolean offer(E o);
    // Inserts the specified element into this queue, if
    // possible.
```

```
  E peek();
    // Retrieves, but does not remove, the head of this queue,
    // returning null if this queue is empty.

  E poll();
    // Retrieves and removes the head of this queue, or null
    // if this queue is empty.

  E remove() throws NoSuchElementException;
    // Retrieves and removes the head of this queue,
    // or throws NoSuchElementException if this queue
    // is empty.
}  // end Queue
```

Note that the *Queue* has one data-type parameter for the items contained in the queue.

Queues are often used to hold elements before processing. In this chapter, the ADT queue that we studied ordered elements in the queue in a FIFO manner. But as you will see later in Chapter 12 when we discuss priority queues, other orderings are possible. Priority queues use a priority level to determine the ordering of the elements in the queue, usually so that the elements with a higher priority will get processed first. Another example is a stack; sometimes you will see a stack referred to as a LIFO queue.

Regardless of the ordering used, the head of the queue is always removed first by a call to *remove* or *poll*. Both methods retrieve and remove an element from the queue, but if the queue is empty, *poll* returns *null*, whereas *remove* will raise *NoSuchElementException*. Similarly, *element* and *peek* can be used to retrieve an element from the queue, with element raising *NoSuchElementException* and *peek* returning *null* if the queue is empty.

Note that *Queue* implementations may allow the insertion of *null* elements. But this makes it difficult to determine if methods such as *poll* and *remove* are returning *null* as an element or *null* meaning that a queue is empty. For this reason, the use of *null* elements is discouraged in queues based on the *Queue* interface.

When adding an element to the queue there are also two methods available: the method *add* (inherited from the interface *Collection*), and the method *offer*. Both methods provide a way to add an element to the queue (based upon the ordering in effect), but differ when adding an element to the queue fails; the add method will return an unchecked exception whereas the *offer* method will return *false*.

The *Queue* interface does not specify the ordering of the elements, it is the responsibility of the class that implements the *Queue* interface to specify its ordering properties through the implementations of the queue interface methods.

The *Deque* Interface. The *Deque* interface extends the *Queue* interface with the following methods (not all of the methods are shown here, see the Java documentation on the *Deque* class for a complete list):

```
public interface Deque<E> extends Queue<E>

  // Three methods for adding an element, if no space is
  // available, they throw IllegalStateException.

  boolean add(E e) throws IllegalStateException;
    // Inserts the specified element into the queue
    // represented by this deque (in other words, at the
    // tail of this deque).

  void addFirst(E e) throws IllegalStateException;
    // Inserts the specified element at the front of this
    // deque.

  void addLast(E e) throws IllegalStateException;
    // Inserts the specified element at the end of this deque.

  // These three methods also add an element, but rather than
  // throw an exception, they return true upon success and
  // false if no space is currently available.

  boolean offer(E e)
    // Inserts the specified element into the queue
    // represented by this deque (in other words, at the
    // tail of this deque), returning true upon success and
    // false if no space is currently available.

  boolean offerFirst(E e)
    // Inserts the specified element at the front of this
    // deque, returning true upon success and false if no
    // space is currently available.

  boolean offerLast(E e)
    // Inserts the specified element at the end of this deque,
    // returning true upon success and false if no space is
    // currently available.

// Three methods for retrieving but not removing an element,
// if the deque is empty, they throw NoSuchElementException.

E element() throws NoSuchElementException;
  // Retrieves, but does not remove, the head of the queue
  // represented by this deque (in other words, the first
  // element of this deque).
```

```
E getFirst() throws NoSuchElementException;
  // Retrieves, but does not remove, the first element of
  // this deque.

E getLast() throws NoSuchElementException;
  // Retrieves, but does not remove, the last element of
  // this deque.

  These three methods also retrieve but do not removing an
  element, but if the deque is empty, they return null.

E peek()
  // Retrieves, but does not remove, the head of the queue
  // represented by this deque (in other words, the first
  // element of this deque).

E peekFirst()
  // Retrieves, but does not remove, the first element of
  // this deque.

E peekLast()
  // Retrieves, but does not remove, the last element of
  // this deque.

// Three methods for retrieving and removing an element,
// if the deque is empty, they throw NoSuchElementException.

E remove() throws NoSuchElementException;
  // Retrieves and removes the head of the queue represented
  // by this deque (in other words, the first element of
  // this deque).

E removeFirst()throws NoSuchElementException;
  // Retrieves and removes the first element of this deque.

E removeLast()throws NoSuchElementException;
  // Retrieves and removes the last element of this deque.

  // These three methods also retrieve and remove an
  // element, but if the deque is empty, they return null.

E poll()
  // Retrieves and removes the head of the queue represented
  // by this deque (in other words, the first element of
  // this deque).
```

```
  E pollFirst()
    // Retrieves and removes the first element of this deque.

  E pollLast()
    // Retrieves and removes the last element of this deque.

    // Miscellaneous methods

  boolean contains(Object o)
    // Returns true if this deque contains the specified
    // element.

  int size()
    // Returns the number of elements in this deque.
} // end Deque
```

Note that, as with the *Queue* interface, many of the methods exist in two forms: one that throws an exception if the operation fails, the other that returns a special value (either *null* or *false*, depending on the operation). We would only expect an insert operation to fail if the implementation is based upon a capacity-restricted data structure, as in our array-based implementation of a queue presented in this chapter. Figure 8-14 and Figure 8-15 summarize the behavior of the various methods for insertions to a deque as well as removal and retrieval from a deque.

Operation	Condition	Throws Exception	No Exception	Return Value
Insert	*Full deque*	`addFirst(e)`	`offerFirst(e)`	***false***
Remove	*Empty deque*	`removeFirst()`	`pollFirst()`	***null***
Examine	*Empty deque*	`getFirst()`	`peekFirst()`	***null***

FIGURE 8-14

Summary of Operations on First Element of Deque

Operation	Condition	Throws Exception	No Exception	Return Value
Insert	*Full deque*	`addLast(e)`	`offerLast(e)`	***false***
Remove	*Empty deque*	`removeLast()`	`pollLast()`	***null***
Examine	*Empty deque*	`getLast()`	`peekLast()`	***null***

FIGURE 8-15

Summary of Operations on Last Element of Deque

Note that a deque can be used as either a stack or a queue depending on the set of methods that are used. When a deque is used as a queue, FIFO (First-In-First-Out) behavior results. A summary of the methods for queues is presented in Figure 8-16. Deques can also be used as LIFO (Last-In-First-Out) stacks. When a deque is used as a stack, elements are pushed and popped from the beginning of the deque. A summary of the methods for stacks is presented in Figure 8-17. Like the JCF `Queue` interface discussed earlier, use of the value `null` is discouraged in deques based on implementations of the `Deque` interface.

`Queue` and `Deque` Implementations. The JCF provides numerous implementations of the `Queue` and `Deque` interfaces. Given that the `Deque` interface is based on the `Queue` interface, there are classes such as `LinkedList` and `ArrayDeque` that implement both interfaces. There are also classes that implement only the `Queue` interface, such as `PriorityQueue`. The `LinkedList` implementation is the one that most closely resembles the queue presented in this chapter. Here is an example of how the JCF `LinkedList` is used as a queue:

```
import java.util.LinkedList;
public class TestQueue {

  static public void main(String[] args) {
    LinkedList<Integer> aQueue = new LinkedList<Integer>();
    boolean ok = true;
    Integer item;
```

ADT Queue	JCF Queue	JCF Deque
`enqueue(e)`	`add(e)`	`addLast(e)`
	`offer(e)`	`offerLast(e)`
`dequeue()`	`remove()`	`removeFirst()`
	`poll()`	`pollFirst()`
`peek()`	`peek()`	`peekFirst()`
	`element()`	`getFirst()`

FIGURE 8-16
Summary of Queue Operations

ADT Stack	JCF Stack	JCF Deque
`push(e)`	`push(e)`	`addFirst(e)`
`pop()`	`pop()`	`removeFirst()`
`peek()`	`peek()`	`peekFirst()`

FIGURE 8-17
Summary of Operations on Last Element of Deque

```
    if (aQueue.isEmpty()) {
      System.out.println("The queue is empty");
    } // end if

    for (int i = 0; i < 5; i++) {
      aQueue.add(i); // With autoboxing, this is the same as
                     // aQueue.add(new Integer(i))
    } // end for

    while (!aQueue.isEmpty()) {
      System.out.print(aQueue.peek()+ " ");
      item = aQueue.remove();
    } // end while
    System.out.println();

  } // end main

} // end TestQueue
```

The output of this program is

```
The queue is empty
0 1 2 3 4
```

Comparing Implementations

We have suggested implementations of the ADT queue that use either a linear linked list, a circular linked list, an array, a circular array, or the ADT list to represent the items in a queue. You have seen the details of three of these implementations. All of our implementations of the ADT queue are ultimately either array based or reference based.

The reasons for making the choice between array-based and reference-based implementations are the same as those discussed in earlier chapters. The discussion here is similar to the one in Chapter 7 in the section "Comparing Implementations." We repeat the highlights here in the context of queues.

Fixed size versus dynamic size

An implementation based on a statically allocated array prevents the *enqueue* operation from adding an item to the queue if the array is full. If this restriction is not acceptable, you must use either a resizeable array or a reference-based implementation.

Reuse of an already implemented class saves you time

Suppose you decide to use a reference-based implementation. Should you choose the implementation that uses a linked list, or should you choose a reference-based implementation of the ADT list? Because a linked list actually represents the items on the ADT list, using the ADT list to represent a queue is not as efficient as using a linked list directly. However, the ADT list approach is much simpler to write.

If you decide to use a linked list instead of the ADT list to represent the queue, should you use a linear linked list or a circular linked list? We leave this question for you to answer in Programming Problem 1.

8.4 A Summary of Position-Oriented ADTs

So far, we have seen three abstract data types—the list, the stack, and the queue—that have a common theme: All of their operations are defined in terms of the positions of their data items. Stacks and queues greatly restrict the positions that their operations can affect; only their end positions can be accessed. The list removes this restriction.

Stacks are really quite similar to queues. This similarity becomes apparent if you pair off their operations, as follows:

Operations for the ADTs list, stack, and queue reference the position of items

A comparison of stack and queue operations

- **`createStack`** and **`createQueue`**. These operations create an empty ADT of the appropriate type.
- Stack **`isEmpty`** and queue **`isEmpty`**. These operations determine whether any items exist in the ADT.
- **`push`** and **`enqueue`**. These operations insert a new item into one end (the top and back, respectively) of the ADT.
- **`pop`** and **`dequeue`**. The *pop* operation deletes the most recent item, which is at the top of the stack, and *dequeue* deletes the first item, which is at the front of the queue.
- Stack **`peek`** and queue **`peek`**. Stack *peek* retrieves the most recent item, which is at the top of the stack, and queue *peek* retrieves the first item, which is at the front of the queue.

The ADT list, introduced in Chapter 4, allows you to insert into, delete from, and inspect the item at any position of the list. Thus, it has the most flexible operations of the three **position-oriented** ADTs. You can view the list operations as general versions of the stack and queue operations, as follows:

ADT list operations generalize stack and queue operations

- **`length`**. If you remove the restriction that the stack and queue versions of `isEmpty` can tell only when an item is present, you obtain an operation that can count the number of items that are present.
- **`add`**. If you remove the restriction that *push* and *enqueue* can insert new items into only one position, you obtain an operation that can insert a new item into any position of the list.
- **`remove`**. If you remove the restriction that *pop* and *dequeue* can delete items from only one position, you obtain an operation that can delete an item from any position of the list.

- **`get`**. If you remove the restriction that the stack and queue versions of **`peek`** can retrieve items from only one position, you obtain an operation that can retrieve the item from any position of the list.

Because each of these three ADTs defines its operations in terms of an item's position in the ADT, this book has presented implementations for them that can provide easy access to specified positions. For example, the stack implementations allow the first position (top) to be accessed quickly, while the queue implementations allow the first position (front) and the last position (back) to be accessed quickly.

8.5 Application: Simulation

Simulation models the behavior of systems

Simulation—a major application area for computers—is a technique for modeling the behavior of both natural and human-made systems. Generally, the goal of a simulation is to generate statistics that summarize the performance of an existing system or to predict the performance of a proposed system. In this section, we will consider a simple example that illustrates one important type of simulation.

Consider the following problem. Ms. Simpson, president of the First City Bank of Springfield, has heard her customers complain about how long they have to wait for service. Because she fears that they may move their accounts to another bank, she is considering whether to hire a second teller.

Before Ms. Simpson hires another teller, she would like an approximation of the average time that a customer has to wait for service from First City's only teller. How can Ms. Simpson obtain this information? She could stand with a stopwatch in the bank's lobby all day, but she does not find this prospect particularly exciting. Besides, she would like to use a method that also allows her to predict how much improvement she could expect if the bank hired a given number of additional tellers. She certainly does not want to hire the tellers on a trial basis and monitor the bank's performance before making a final decision.

Ms. Simpson concludes that the best way to obtain the information she wants is to use a computer model to simulate the behavior of her bank. The first step in simulating a system such as a bank is to construct a mathematical model that captures the relevant information about the system. For example, how many tellers does the bank employ? How often do customers arrive? If the model accurately describes the real-world system, a simulation can derive accurate predictions about the system's overall performance. For example, a simulation could predict the average time a customer has to wait before receiving service. A simulation can also evaluate proposed changes to the real-world system. For example, it could predict the effect of hiring more tellers in the bank. A large decrease in the time predicted for the average wait of a customer might justify the cost of hiring additional tellers.

Simulated time

Central to a simulation is the concept of simulated time. Envision a stopwatch that measures time elapsed during a simulation. For example, suppose

that the model of the bank specifies only one teller. At time 0, which is the start of the banking day, the simulated system would be in its initial state with no customers. As the simulation runs, the stopwatch ticks away units of time—perhaps minutes—and certain events occur. At time 12, the bank's first customer arrives. Since there is no line, the customer goes directly to the teller and begins her transaction. At time 20, a second customer arrives. Because the first customer has not yet completed her transaction, the second customer must wait in line. At time 38, the first customer completes her transaction and the second customer can begin his. Figure 8-18 illustrates these four times in the simulation.

To gather the information you need, you run this simulation for a specified period of simulated time. During the course of the run, you need to keep track of certain statistics, such as the average time a customer has to wait for service. Notice that in the small example of Figure 8-18, the first customer had to wait 0 minutes to begin a transaction and the second customer had to wait 18 minutes to begin a transaction—an average wait of 9 minutes.

One point not addressed in the previous discussion is how to determine when certain events occur. For example, why did we say that the first customer arrived at time 12 and the second at time 20? By studying real-world systems like our bank, mathematicians have learned to model events such as the arrival of people, using techniques from probability theory. This statistical information is incorporated into the mathematical model of the system and is used to generate events in a way that reflects the real world. The simulation uses these events and is thus called an **event-driven simulation.** Note that the goal is to reflect the long-term average behavior of the system rather than to predict occurrences of specific events. This goal is sufficient for the needs of the simulation.

Although the techniques for generating events to reflect the real world are interesting and important, they require a good deal of mathematical sophistication. Therefore, simply assume that you already have a list of events available for your use. In particular, for the bank problem, assume that a file contains the time of each customer's arrival—an **arrival event**—and the duration of that customer's transaction once the customer reaches the teller. For example, the data

Sample arrival and transaction times

```
20    5
22    4
23    2
30    3
```

indicates that the first customer arrives 20 minutes into the simulation and that the transaction—once begun—requires 5 minutes; the second customer arrives 22 minutes into the simulation and the transaction requires 4 minutes; and so on. Assume that the input file is ordered by arrival time.

Notice that the file does not contain **departure events**; the data does not specify when a customer will complete the transaction and leave. Instead, the simulation must determine when departures occur. By using the arrival time

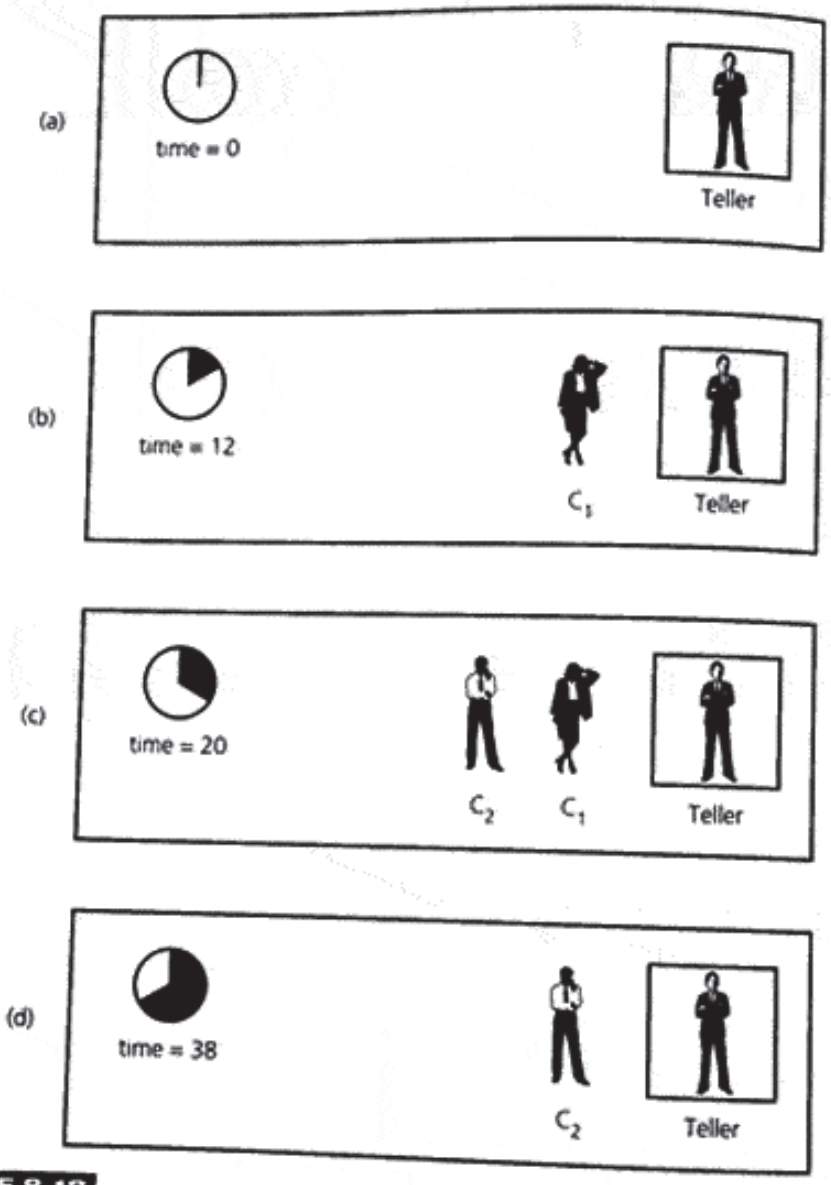


FIGURE 8-18

A bank line at time (a) 0; (b) 12; (c) 20; (d) 38

and the transaction length, the simulation can easily determine the time at which a customer departs. To see how to make this determination, you can conduct a simulation by hand with the previous data as follows:

The results of a simulation

Time	Event
20	Customer 1 enters bank and begins transaction
22	Customer 2 enters bank and stands at end of line

```
23      Customer 3 enters bank and stands at end of line
25      Customer 1 departs; customer 2 begins transaction
29      Customer 2 departs; customer 3 begins transaction
30      Customer 4 enters bank and stands at end of line
31      Customer 3 departs; customer 4 begins transaction
34      Customer 4 departs
```

A customer's wait time is the elapsed time between arrival in the bank and the start of the transaction. The average of this wait time over all the customers is the statistic that you want to obtain.

To summarize, this simulation is concerned with two types of events:

- **Arrival events.** These events indicate the arrival at the bank of a new customer. The input file specifies the times at which the arrival events occur. As such, they are **external events.** When a customer arrives at the bank, one of two things happens. If the teller is idle when the customer arrives, the customer enters the line and begins the transaction immediately. If the teller is busy, the new customer must stand at the end of the line and wait for service.
- **Departure events.** These events indicate the departure from the bank of a customer who has completed a transaction. The simulation determines the times at which the departure events occur. As such, they are **internal events.** When a customer completes the transaction, he or she departs and the next person in line—if there is one—begins a transaction.

The main tasks of an algorithm that performs the simulation are to determine the times at which the events occur and to process the events when they do occur. The algorithm is stated at a high level as follows:

A first attempt at a simulation algorithm

```
// initialize
currentTime = 0
Initialize the line to "no customers"

while (currentTime ≤ time of the final event) {

  if (an arrival event occurs at time currentTime) {
    Process the arrival event
  }  // end if
  if (a departure event occurs at time currentTime) {
    Process the departure event
  }  // end if
  // when an arrival event and departure event
  // occur at the same time, arbitrarily process
  // the arrival event first
```

```
    ++currentTime
  }  // end while
```

A time-driven simulation simulates the ticking of a clock

An event-driven simulation considers only times of certain events, in this case, arrivals and departures

But do you really want to increment *currentTime* by 1? You would for a **time-driven simulation,** where you would determine arrival and departure times at random and compare those times to *currentTime*. In such a case, you would increment *currentTime* by 1 to simulate the ticking of a clock. Recall, however, that this simulation is event driven, so you have a file of arrival times and transaction times. Because you are interested only in those times at which arrival and departure events occur and because no action is required between events, you can advance *currentTime* from the time of one event directly to the time of the next.

Thus, you can revise the pseudocode solution as follows:

First revision of the simulation algorithm

```
// initialize the line to "no customers"

while (events remain to be processed) {
  currentTime = time of next event
  if (event is an arrival event) {
    Process the arrival event
  }
  else {
    Process the departure event
  }  // end if
  // when an arrival event and departure event
  // occur at the same time, arbitrarily process
  // the arrival event first
}  // end while
```

You must determine the time of the next arrival or departure event so that you can implement the statement

```
currentTime = time of next event
```

An event list contains all future events

To make this determination, you must maintain an **event list.** An event list contains all arrival and departure events that will occur but have not occurred yet. The times of the events in the event list are in ascending order, and thus the next event to be processed is always at the beginning of the list. The algorithm simply gets the event from the beginning of the list, advances to the time specified, and processes the event. The difficulty, then, lies in successfully managing the event list.

Since each arrival event generates exactly one departure event, you might think that you should read the entire input file and create an event list of all arrival and departure events sorted by time. Self-Test Exercise 5 asks you to explain why this approach is impractical. As you will see, you can instead

manage the event list for this particular problem so that it always contains at most one event of each kind.

This event list contains at most one arrival event and one departure event

Recall that the arrival events are specified in the input file in ascending time order. You thus never need to worry about an arrival event until you have processed all the arrival events that precede it in the file. You simply keep the earliest unprocessed arrival event in the event list. When you eventually process this event—that is, when it is time for this customer to arrive—you replace it in the event list with the next unprocessed arrival event, which is the next item in the input file.

Similarly, you need to place only the next departure event to occur on the event list. But how can you determine the times for the departure events? Observe that the next departure event always corresponds to the customer that the teller is currently serving. As soon as a customer begins service, the time of his or her departure is simply

time of next departure = time service begins + length of transaction

Recall that the length of the customer's transaction is in the input file, along with the arrival time. Thus, as soon as a customer begins service, you place a departure event corresponding to this customer in the event list. Figure 8-19 illustrates a typical instance of the event list for this simulation.

Now consider how you can process an event when it is time for the event to occur. You must perform two general types of actions:

- **Update the line:** Add or remove customers.
- **Update the event list:** Add or remove events.

Two tasks are required to process each event

As customers arrive, they go to the back of the line. The current customer, who is at the front of the line, is being served, and it is this customer that you remove from the system next. It is thus natural to use a queue to represent the line of customers in the bank. For this problem, the only information that you

A queue represents the customers in line

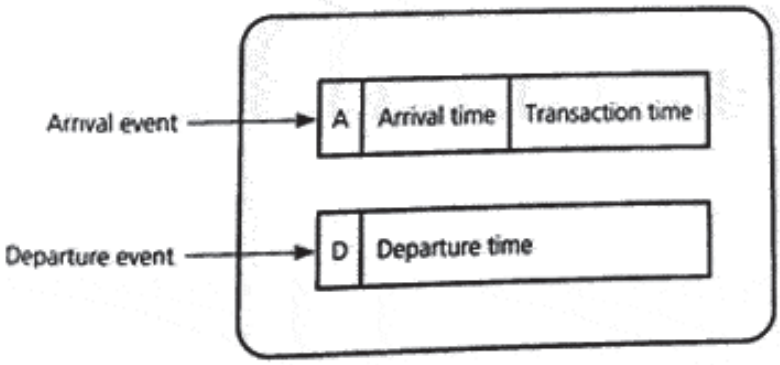


FIGURE 8-19

A typical instance of the event list

The event list is not a queue

must store in the queue about each customer is the time of arrival and the length of the transaction. The event list, since it is sorted by time, is not a queue. We will examine it in more detail shortly.

To summarize, you process an event as follows:

The algorithm for arrival events

```
TO PROCESS AN ARRIVAL EVENT

  // Update the event list
  Delete the arrival event for customer C from
      the event list

  if (new customer C begins transaction immediately) {
    Insert a departure event for customer C into the
        event list (time of event = current time +
        transaction length)
  }  // end if
  if (not at the end of the input file) {
    Read a new arrival event and add it to the event list
        (time of event = time specified in file)
  }  // end if
```

A new customer always enters the queue and is served while at the queue's front

Because a customer is served while at the front of the queue, a new customer always enters the queue, even if the queue is empty. You then delete the arrival event for the new customer from the event list. If the new customer is served immediately, you insert a departure event into the event list. Finally, you read a new arrival event into the event list. This arrival event can occur either before or after the departure event.

The algorithm for departure events

```
TO PROCESS A DEPARTURE EVENT

  // Update the line
  Delete the customer at the front of the queue
  if (the queue is not empty) {
     The current front customer begins transaction
  }  // end if

  // Update the event list
  Delete the departure event from the event list
  if (the queue is not empty) {
     Insert into the event list the departure event for
       the customer now at the front of the queue
       (time of event = current time + transaction length)
  }  // end if
```

After processing the departure event, you do not read another arrival event from the file. Assuming that the file has not been read completely, the event

list will contain an arrival event whose time is earlier than any arrival still in the input file.

Examining the event list more closely will help explain the workings of the algorithm. There is no typical form that an event list takes. For this simulation, however, the event list has four possible configurations:

Four configurations of the event list for this simulation

- Initially, the event list contains an arrival event A after you read the first arrival event from the input file but before you process it:

  Event list: A (initial state)

- Generally, the event list for this simulation contains exactly two events: one arrival event A and one departure event D. Either the departure event is first or the arrival event is first as follows:

  Event list: D A (general case—next event is a departure)

  or

  Event list: A D (general case—next event is an arrival)

- If the departure event is first and that event leaves the teller's line empty, a new departure event does not replace the just-processed event. Thus, in this case, the event list appears as

  Event list: A (a departure leaves the teller's line empty)

  Notice that this instance of the event list is the same as its initial state.

- If the arrival event is first and if, after it is processed, you are at the end of the input file, the event list contains only a departure event:

  Event list: D (the input has been exhausted)

Other situations result in an event list that has one of the previous four configurations.

You insert new events either at the beginning of the event list or at the end, depending on the relative times of the new event and the event currently in the event list. For example, suppose that the event list contains only an arrival event A and that another customer is now at the front of the line and beginning a transaction. You need to generate a departure event D for this customer. If the customer's departure time is before the time of the arrival event A, you must insert the departure event D before the event A in the event list. However, if the departure time is after the time of the arrival event, you must insert the departure event D after the arrival event A. In the case of a tie, you need a rule to determine which event should take precedence. In this solution, we arbitrarily choose to place the departure event after the arrival event.

You can now combine and refine the pieces of the solution into an algorithm that performs the simulation by using the ADT queue operations to manage the bank line:

The final pseudocode for the event-driven simulation

```
+simulate()
// Performs the simulation.

  Create an empty queue bankQueue to represent the bank line
  Create an empty event list eventList

  Get the first arrival event from the input file
  Place the arrival event in the event list

  while (the event list is not empty) {
    newEvent = the first event in the event list

    if (newEvent is an arrival event) {
      processArrival(newEvent, arrivalFile,
                     eventList, bankQueue)
    }
    else {
      processDeparture(newEvent, eventList, bankQueue)
    }  // end if
  }  // end while

+processArrival(in arrivalEvent:Event,
                in arrivalFile:File,
                inout anEventList:EventList,
                inout bankQueue:Queue)
// Processes an arrival event.

  atFront = bankQueue.isEmpty()  // present queue status

  // update the bankQueue by inserting the customer, as
  // described in arrivalEvent, into the queue
  bankQueue.enqueue(arrivalEvent)

  // update the event list
  Delete arrivalEvent from anEventList

  if (atFront) {
    // the line was empty, so new customer is at front
    // of line and begins transaction immediately
    Insert into the anEventList a departure event that
      corresponds to the new customer and has
    currentTime = currentTime + transaction length
  }  // end if

  if (not at end of input file)
    Get the next arrival event from arrivalFile
    Insert event with time specified in it
```

```
      file -- to anEventList
    end if

processDeparture(in departureEvent:Event,
                 in anEventList:EventList,
                 inout bankQueue:Queue)
processes a departure event.

  update the line by deleting the front customer
bankQueue.dequeue()

  update the event list
delete departureEvent from anEventList
if (!bankQueue.isEmpty())
    customer at front of line begins transaction
  insert into anEventList a departure event that
    corresponds to the customer now at the front of the
    line and has currentTime = currentTime
    + transaction length
} // end if
```

Figure 8-20 begins a trace of this algorithm for the data on page 436 and shows the changes to the queue and event list. Self-Test Exercise 6 at the end of this chapter asks you to complete the trace.

The event list is, in fact, an ADT. By examining the previous pseudocode, you can see that this ADT must include at least the following operations:

ADT event list operations

```
+createEventList()
// Creates an empty event list.

+isEmpty():boolean {query}
// Determines whether an event list is empty.

+insert(in anEvent:Event)
// Inserts anEvent into an event list so that events
// are ordered by time. If an arrival event and a
// departure event have the same time, the arrival
// event precedes the departure event.

+delete()
// Deletes the first event from an event list.

+retrieve():Event
// Retrieves the first event in an event list.
```

Time	Action	bankQueue (front to back)	anEventList (beginning to end)
0	Read file, place event in anEventList	(empty)	A 20 5
20	Update anEventList and bankQueue Customer 1 enters bank	20 5	(empty)
	Customer 1 begins transaction, create departure event	20 5	D 25
	Read file, place event in anEventList	20 5	A 22 4 D 25
22	Update anEventList and bankQueue Customer 2 enters bank	20 5 22 4	D 25
	Read file, place event in anEventList	20 5 22 4	A 23 2 D 25
23	Update anEventList and bankQueue Customer 3 enters bank	20 5 22 4 23 2	D 25
	Read file, place event in anEventList	20 5 22 4 23 2	D 25 A 30 3
25	Update anEventList and bankQueue Customer 1 departs	22 4 23 2	A 30 3
	Customer 2 begins transaction, create departure event	22 4 23 2	D 29 A 30 3

Self-Test Exercise 6 asks you to complete this trace

FIGURE 8-20

A partial trace of the bank simulation algorithm for the data
20 5
22 4
23 2
30 3

Programming Problem 8 at the end of this chapter asks you to complete the implementation of this simulation.

Summary

1. The definition of the queue operations gives the ADT queue first-in, first-out (FIFO) behavior.
2. The insertion and deletion operations for a queue require efficient access to both ends of the queue. Therefore, a reference-based implementation of a queue uses either a circular linked list or a linear linked list that has both a head reference and a tail reference.
3. An array-based implementation of a queue is prone to rightward drift. This phenomenon can make a queue look full when it really is not. Shifting the items in the array is one way to compensate for rightward drift. A more efficient solution uses a circular array.
4. If you use a circular array to implement a queue, you must be able to distinguish between the queue-full and queue-empty conditions. You can make this distinction

by either counting the number of items in the queue, using a `full` flag, or leaving one array location empty.

5. Models of real-world systems often use queues. The event-driven simulation in this chapter uses a queue to model a line of customers in a bank.

6. Central to a simulation is the notion of simulated time. In a time-driven simulation, simulated time is advanced by a single time unit, whereas in an event-driven simulation, simulated time is advanced to the time of the next event. To implement an event-driven simulation, you maintain an event list that contains events that have not yet occurred. The event list is ordered by the time of the events so that the next event to occur is always at the head of the list.

Cautions

1. If you use a linear linked list with only a head reference to implement a queue, the insertion operation will be inefficient. Each insertion requires a traversal to the end of the linked list. As the queue increases in length, the traversal time—and hence the insertion time—will increase.
2. The management of an event list in an event-driven simulation is typically more difficult than it was in the example presented in this chapter. For instance, if the bank had more than one teller line, the structure of the event list would be much more complex.

Self-Test Exercises

1. If you add the letters W, Y, X, Z, and V in sequence to a queue of characters and then remove them, in what order will they be deleted from the queue?
2. What do the initially empty queues *queue1* and *queue2* "look like" after the following sequence of operations?

```
queue1.enqueue(23)
queue1.enqueue(17)
queue1.enqueue(50)
queue2.enqueue(42)
qFront1 = queue1.dequeue()
qFront2 = queue2.peek()
queue2.enqueue(top1)
queue1.enqueue(top2)
queue1.enqueue(13)
qFront2 = queue2.pop()
queue2.enqueue(49)
```

   Compare these results with Self-Test Exercise 2 in Chapter 7.

3. Trace the palindrome-recognition algorithm described in the section "Simple Applications of the ADT Queue" for each of the following strings:

   a. abracadabra
   b. radar
   c. rotator
   d. xyzzy

4. For each of the following situations, which of these ADTs (1 through 4) would be most appropriate: (1) a queue; (2) a stack; (3) a list; (4) none of these?

   a. The customers at a deli counter who take numbers to mark their turn

   b. An alphabetic list of names

   c. Integers that need to be sorted

   d. The boxes in a box trace of a recursive method

   e. A grocery list ordered by the occurrence of the items in the store

   f. The items on a cash register tape

   g. A word processor that allows you to correct typing errors by using the backspace key

   h. A program that uses backtracking

   i. A list of ideas in chronological order

   j. Airplanes that stack above a busy airport, waiting to land

   k. People who are put on hold when they call an airline to make reservations

   l. An employer who fires the most recently hired person

   m. People who go to a store on Black Friday hoping to get the best holiday buys

5. In the bank simulation problem that this chapter discusses, why is it impractical to read the entire input file and create a list of all the arrival and departure events before the simulation begins?

6. Complete the hand trace of the bank-line simulation that Figure 8-20 began with the data given on page 436. Show the state of the queue and the event list at each step.

Exercises

1. What makes a linked list a good choice for implementing a queue? Is an array an equally good choice? Explain your answer.

2. Compare the operations in the ADT stack with the ADT queue. Which operations are basically the same? What operations differ? Explain your answer.

3. Suppose you have a queue in which the values 1 through 5 must be enqueued on the queue in that order, but that an item on the queue can be dequeued and printed at any time. Based on these constraints, give the list of operations that would produce each of the following sequences. If it is not possible, state so.

   a. 1 3 5 4 2

   b. 1 2 3 4 5

   c. Are there sequences that cannot occur? Explain why or why not.

4. Implement the pseudocode conversion algorithm that converts a sequence of character digits in a queue to an integer (it is in the section "Reading a String of Characters"). Assume that you are using a queue to read in a series of characters that represent a correct postfix expression. The postfix expression has operators and multi-digit integers separated by single blanks. When the conversion method is called, the next item in the queue should be a character digit followed by zero or more character digits. The digits should be read until a non-digit character is found, and the resulting integer returned.

5. Consider the palindrome-recognition algorithm described in the section "Simple Applications of the ADT Queue." Is it necessary for the algorithm to look at the entire queue and stack? That is, can you reduce the number of times that the loop must execute?

6. Consider the language

$$L = \{w\$w' \mid w \text{ is a possibly empty string of characters other than \$},\ w' = reverse(w)\}$$

as defined in Chapter 7. Write a recognition algorithm for this language that uses both a queue and a stack. Thus, as you traverse the input string, you insert each character of w into a queue and each character of w' into a stack. Assume that each input string contains exactly one \$.

7. What is output by the following code section?

```
QueueInterface aQueue = new QueueReferenceBased();
int num1, num2;
for (int i = 1; i <= 5; i++) {
  aQueue.enqueue(i);
} // end for

for (int i = 1; i <= 5; i++) {
  num1 = (Integer)aQueue.dequeue();
  num2 = (Integer)aQueue.dequeue();
  aQueue.enqueue(num1 + num2);
  aQueue.enqueue(num2 - num1);
} // end for

while(!aQueue.isEmpty()) {
  System.out.print(aQueue.dequeue() + " ");
} // end for
```

8. Assume you have a queue q that has already been populated with data. What does the following code fragment do to the queue q?

```
Stack s = new Stack();
while (!q.isEmpty())
   s.push(q.dequeue());
while (!ss.isEmpty())
   q.enqueue(s.pop());
```

9. Another operation that could be added to the ADT Queue is one that removes and discards the user-specified number of elements from the front of the queue.

Assume this operation is called *dequeueAndDiscard* and that it does not return a value and accepts a parameter called *count* of data type `int`.

a. Add this operation to the *QueueListBased* implementation given in this chapter.

b. Add this operation to the *QueueArrayBased* implementation given in this chapter.

c. Add this operation to the *QueueReferenceBased* implementation given in this chapter.

10. The JCF class *Deque* had a method called *contains* that would return true if an *Object* was in the deque. For a queue, the method could be specified as follows:

```
public boolean contains(Object o)
// Returns true if this queue contains the specified element.
```

a. Add this method to the *QueueListBased* implementation given in this chapter.

b. Add this method to the *QueueArrayBased* implementation given in this chapter.

c. Add this method to the *QueueReferenceBased* implementation given in this chapter.

11. Revise the infix-to-postfix conversion algorithm of Chapter 7 so that it uses a queue to represent the postfix expression.

12. Consider the queue implementation that uses the ADT list to represent the items in the queue. Discuss the efficiency of the queue's insertion and deletion operations when the ADT list's implementation is

a. Array based

b. Reference based

13. An operation that displays the contents of a queue can be useful during program debugging. Add a *display* operation to the ADT queue such that

a. *display* uses only ADT queue operations, so it is independent of the queue's implementation

b. *display* assumes and uses the reference-based implementation of the ADT queue

14. Write a client method that returns the last element of a queue, while leaving the queue unchanged. This method can call any of the methods of the queue interface. It can also declare new queue objects. The return type of the method is *Object* and the method accepts a queue object (either a *QueueArrayBased* object or a *QueueListBased* object) as a parameter.

15. The Java Collections Framework provides a class called *ArrayDeque* that is an implementation of the *Deque* interface presented in this chapter. Use the class *ArrayDeque* to solve the read-and-correct problem presented in the "Developing an ADT During the Design of a Solution" section of Chapter 7. In that problem, you enter text at a keyboard and correct typing mistakes by using the backspace key. Each backspace erases the most recently entered character. Your solution should provide a corrected string of characters in the order in which they were entered at the keyboard.

16. With the following data, hand-trace the execution of the bank-line simulation that this chapter describes. Each line of data contains an arrival time and a transaction time. Show the state of the queue and the event list at each step.

5	5
-	9
8	4
23	6
30	5
33	4
38	6

Note that at time 23 there is a tie between the execution of an arrival event and a departure event.

17. In the solution to the bank simulation problem, can the event list be an ADT queue? Can the event list be an ADT list or sorted list?

18. Consider the stack-based search of the flight map in the HPAir problem of Chapter 7. You can replace the stack that `searchS` uses with a queue. That is, you can replace every call to *push* with a call to *enqueue*, every call to pop with a call to *dequeue*, and every call to the stack version of *peek* with a call to the queue version of *peek*. Trace the resulting algorithm when you fly from *P* to *Z* in the flight map in Figure 7-10. Indicate the contents of the queue after every operation on it.

19. As Chapter 4 pointed out, you can define ADT operations in a mathematically formal way by using axioms. Consider the following axioms for the ADT queue, where *queue* is an arbitrary queue and `item` is an arbitrary queue item.

```
(queue.createQueue()).isEmpty() = true
  (queue.enqueue(item)).isEmpty() = false
(queue.createQueue()).dequeue() = error
((queue.createQueue()).enqueue(item)).dequeue() =
                                    queue.createQueue()
queue.isEmpty() = false ⇒
    (queue.enqueue(item)).dequeue() =
                        (queue.dequeue()).enqueue(item)
(queue.createQueue()).peek() = error
((queue.createQueue()).enqueue(item)).peek() = item
queue.isEmpty() = false ⇒
    (queue.enqueue(item)).peek() = queue.peek()
```

a. Note the recursive nature of the definition of *peek*. What is the base case? What is the recursive step? What is the significance of the `isEmpty` test? Why is *peek* recursive in nature while the operation *peek* for the ADT stack is not?

b. The representation of a stack as a sequence of *push* operations without any *pop* operations was called a canonical form. (See Exercise 16a in Chapter 7.) Is there a canonical form for the ADT queue that uses only *enqueue* operations? That is, is every queue equal to a queue that can be written with only *enqueues*? Prove your answer.

Programming Problems

1. Write a reference-based implementation of a queue that uses a linear linked list to represent the items in the queue. You will need both a head reference and a tail reference. When you are done, compare your implementation to the one given in this chapter that uses a circular linked list with one external reference. Which implementation is easier to write? Which is easier to understand? Which is more efficient?

2. The array based implementation of the ADT queue in this chapter assumed a maximum queue size of 50 items. Modify this implementation so that when the queue becomes full, the size of the array is doubled.

3. Consider the array-based implementation of a queue given in the text. Instead of counting the number of items in the queue, you could maintain a flag `full` to distinguish between the full and empty conditions. Revise the array-based implementation by using the `full` flag.

   a. Does this implementation have the same space requirements as the `count` or `full` implementations? Why?

   b. Implement this array-based approach.

4. This chapter described another array-based implementation of a queue that uses no special data field—such as `count` or `full` (see Programming Problem 3)—to distinguish between the full and empty conditions. In this implementation, you declare `MAX_QUEUE + 1` locations for the array `items`, but use only `MAX_QUEUE` of them for queue items. You sacrifice one array location by making `front` the index of the location before the front of the queue. The queue is full if `front` equals `(back+1) % (MAX_QUEUE+1)`, but the queue is empty if `front` equals `back`.

   a. Does this implementation have the same space requirements as the `count` or `full` implementations? Why?

   b. Implement this array-based approach.

5. Implement the palindrome-recognition algorithm described in the section "Simple Applications of the ADT Queue."

6. Implement the recognition algorithm described in Exercise 5 using the JCF `Stack` and `LinkedList` classes.

7. As discussed in this chapter, the JCF provides an interface for a double ended queue that supports insertion and deletion of items from both the front and back of the data structure. Here is a simplified version of a deque interface:

```
public interface Deque {

  public boolean isEmpty();
    // Return true if the deque is empty, false otherwise.

  public boolean addFirst(Object item);
    // Insert the item at the front of the deque. Returns false
    // if the item cannot be added to the deque, true otherwise.
```

```
public boolean addLast(Object item);
     Insert the item at the back of the deque. Returns false if
     the item cannot be added to the deque, true otherwise.

public Object removeFirst();
     Delete and return the first item in the deque if the deque
     is not empty, otherwise return null (the deque was empty).

public Object removeLast();
     Delete and return the last item in the deque if the deque
     is not empty, otherwise return null (the deque was empty).

public Object peekFirst();
     Return the first item in the deque if the deque is
     not empty, leaving the deque unchanged. Otherwise return
     null (the deque was empty).

public Object peekLast();
     Return the last item in the deque if the deque is
     not empty, leaving the deque unchanged. Otherwise return
     null (the deque was empty).
end Deque
```

4. Create a reference-based implementation of the *Deque* interface.

5. Create an array-based implementation of the *Deque* interface.

8. Implement the event-driven simulation of a bank that this chapter described. A queue of arrival events will represent the line of customers in the bank. Maintain the arrival events and departure events in an ADT event list, sorted by the time of the event. Use a reference-based implementation for the ADT event list.

   The input is a text file of arrival and transaction times. Each line of the file contains the arrival time and required transaction time for a customer. The arrival times are ordered by increasing time.

   Your program must count customers and keep track of their cumulative waiting time. These statistics are sufficient to compute the average waiting time after the last event has been processed.

   Display a trace of the events executed and a summary of the computed statistics (total number of arrivals and average time spent waiting in line). For example, the input file shown in the left columns of the following table should produce the output shown in the right column.

Input File		Output
1	5	Simulation Begins
2	5	Processing an arrival event at time: 1
4	5	Processing an arrival event at time: 2
20	5	Processing an arrival event at time: 4
22	5	Processing a departure event at time: 6
24	5	Processing a departure event at time: 11
26	5	Processing a departure event at time: 16
28	5	Processing an arrival event at time: 20
30	5	Processing an arrival event at time: 22
88	3	Processing an arrival event at time: 24
		Processing a departure event at time: 25

```
Processing an arrival event at time:   26
Processing an arrival event at time:   28
Processing an arrival event at time:   30
Processing a departure event at time:  30
Processing a departure event at time:  35
Processing a departure event at time:  40
Processing a departure event at time:  45
Processing a departure event at time:  50
Processing an arrival event at time:   88
Processing a departure event at time:  91
Simulation Ends

Final Statistics:
Total number of people processed: 10
Average amount of time spent waiting: 5.6
```

9. Modify and expand the event-driven simulation program that you wrote in Programming Problem 6. Here are a few suggestions:

   a. Add an operation that displays the event list, and use it to check your hand trace in Exercise 10.

   b. Add some statistics to the simulation. For example, compute the maximum wait in line, the average length of the line, and the maximum length of the line.

   c. Modify the simulation so that it accounts for three tellers, each with a distinct line. You should keep in mind that there should be

      - Three queues, one for each teller
      - A rule that chooses a line when processing an arrival event (for example, enter the shortest line)
      - Three distinct departure events, one for each line
      - Rules for breaking ties in the event list

      Run both this simulation and the original simulation on several sets of input data. How do the statistics compare?

   d. The bank is considering the following change: Instead of having three distinct lines (one for each teller), there will be a single line for the three tellers. The person at the front of the line will go to the first available teller. Modify the simulation of Part *c* to account for this variation. Run both simulations on several sets of input data. How do the various statistics compare (averages and maximums)? What can you conclude about having a single line as opposed to having distinct lines?

10. The people that run the Motor Vehicle Department (MVD) have a problem. They are concerned that people do not spend enough time waiting in lines to appreciate the privilege of owning and driving an automobile. The current arrangement is as follows:

    - When people walk in the door, they must wait in a line to sign in.
    - Once they have signed in, they are told either to stand in line for registration renewal or to wait until they are called for license renewal.
    - Once they have completed their desired transaction, they must go and wait in line for the cashier.

- When they finally get to the front of the cashier's line, if they expect to pay by check, they are told that all checks must get approved. To do this, it is necessary to go to the check-approver's table and then reenter the cashier's line at the end.

Write an event-driven simulation to help the Motor Vehicle Department gather statistics.

Each line of input will contain

- A desired transaction code (L for license renewal, R for registration renewal)
- A method-of-payment code (S for cash, C for check)
- An arrival time (integer)
- A name

Write out the specifics of each event (when, who, what, and so on). Then display these final statistics:

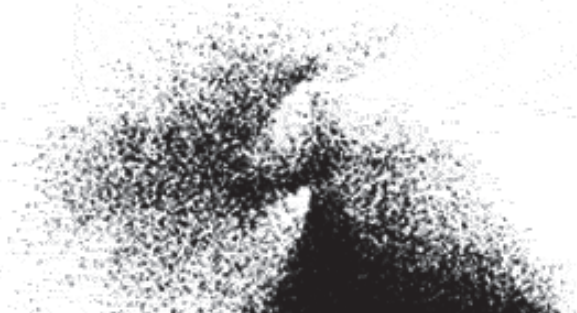
- The total number of license renewals and the average time spent in MVD (arrival until completion of payment) to renew a license
- The total number of registration renewals and the average time spent in MVD (arrival until completion of payment) to renew a registration

Incorporate the following details into your program:

- Define the following events: arrive, sign in, renew license, renew registration, and interact with the cashier (make a payment or find out about check approval).
- In the case of a tie, let the order of events be determined by the list of events just given—that is, arrivals have the highest priority.
- Assume that the various transactions take the following amounts of time:

Sign in	10 seconds
Renew license	90 seconds
Register automobile	60 seconds
See cashier (payment)	30 seconds
See cashier (check not approved)	10 seconds

- As ridiculous as it may seem, the people waiting for license renewal are called in alphabetical order. Note, however, that people are not pushed back once their transactions have started.
- For the sake of this simulation, you can assume that checks are approved instantly. Therefore, the rule for arriving at the front of the cashier's line with a check that has not been approved is to go to the back of the cashier's line with a check that has been approved.



CHAPTER 9

Advanced Java Topics

Java classes provide a way to enforce the walls of data abstraction by encapsulating an abstract data type's data and operations. An object-oriented approach, however, goes well beyond encapsulation. Inheritance and polymorphism allow you to derive new classes from existing classes. This chapter describes techniques that make collections of reusable software components possible. It also discusses some of the useful components that exist in the Java API and how they can be used. Realize that much more can and should be said about these techniques. Consider this chapter as an introduction to this material.

9.1 Inheritance Revisited

When you think of inheritance, you might imagine a bequest of one million dollars from some long-lost wealthy relative. In the object-oriented world, however, **inheritance** describes the ability of a class to derive properties from a previously defined class. These properties are like the genetic characteristics you received from your parents: Some traits are the same, some are similar but different, and some are new.

A class can derive the behavior and structure of another class

Inheritance, in fact, is a relationship among classes. One class can derive the behavior and structure of another class. For example, Figure 9-1 illustrates some relationships among various timepieces. Digital clocks, for example, include the clock in the dashboard of your car, the clock on the sign of the downtown bank, and the clock on your microwave oven. All digital clocks have the same underlying structure and perform operations such as

```
Set the time
Advance the time
Display the time
```

A digital alarm clock is a digital clock

A digital alarm clock is a digital clock that also has alarm methods, such as

```
Set the alarm
Enable the alarm
Sound the alarm
Silence the alarm
```

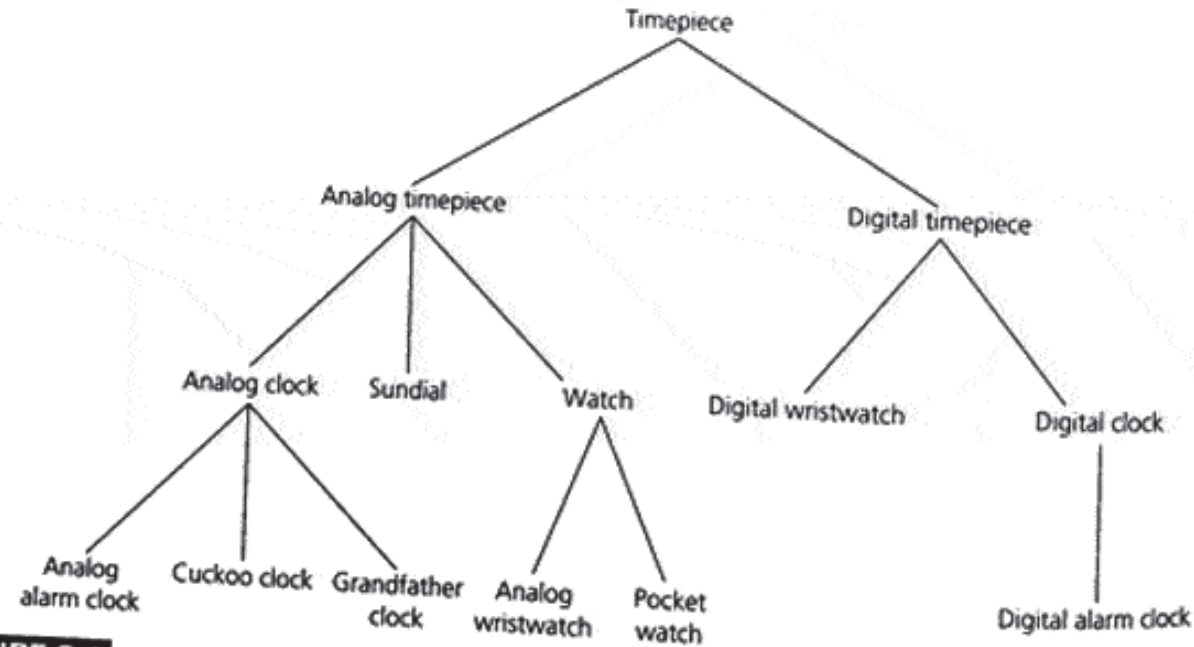


FIGURE 9-1
Inheritance: Relationships among timepieces

That is, a digital alarm clock has the structure and operations of a digital clock and, in addition, has an alarm and operations to manipulate the alarm.

You can think of the group of digital clocks and the group of digital alarm clocks as classes. The class of digital alarm clocks is a **subclass** or a **derived class** of the class of digital clocks. The class of digital clocks is a **superclass** or **base class** of the class of digital alarm clocks.

A subclass **inherits** all the members of its superclass, except the constructors. That is, a subclass has the data fields and methods of the superclass in addition to the data fields and methods it defines. A subclass can also have its own version of an inherited method. For example, according to Figure 9-1, a cuckoo clock is a **descendant,** or subclass, of an analog clock, like the one on a classroom wall. The cuckoo clock inherits the structure and behavior of the analog clock, but revises the way it reports the time each hour by adding a cuckoo.

A subclass inherits the members of its superclass

Inheritance enables you to **reuse** software components when you define a new class. For example, you can reuse your design and implementation of an analog clock when you design a cuckoo clock. A simpler example will demonstrate the details of such reuse and show you how Java implements inheritance.

Inheritance enables the reuse of existing classes

Chapter 4 spoke of volleyballs and soccer balls as objects. While designing a class of balls—*Ball*—you might decide that a ball is simply a sphere with a name. This realization is significant in that *Sphere*—the class of spheres—already exists. Thus, if you let *Sphere* be a superclass of *Ball*, you can implement *Ball* without reinventing the sphere. Toward that end, here is a definition of *Sphere* that is similar to the *SimpleSphere* class presented in Chapter 1:

Inheritance reduces the effort necessary to add features to an existing object

```
public class Sphere {
 private double radius;
 public static final double DEFAULT_RADIUS = 1.0;

 public Sphere() {
   setRadius(DEFAULT_RADIUS);
 } // end default constructor

 public Sphere(double initialRadius) {
   setRadius(initialRadius);
 } // end constructor

 public boolean equals(Object rhs) {
   return ((rhs instanceof Sphere) &&
           (radius == ((Sphere)rhs).radius));
 }  // end equals

 Public void setRadius(double newRadius) {
   if (newRadius >= 0.0) {
     radius = newRadius;
   }  // end if
 } // end setRadius
```

```
  public double getRadius() {
    return radius;
  } // end getRadius

  public double diameter() {
    return 2.0 * radius;
  } // end diameter

  public double circumference() {
    return Math.PI * diameter();
  } // end circumference

  public double area() {
    return 4.0 * Math.PI * radius * radius;
  } // end area

  public double volume() {
    return (4.0*Math.PI * Math.pow(radius, 3.0)) / 3.0;
  } // end volume

  public void displayStatistics() {
    System.out.println("\nRadius = " + getRadius() +
                       "\nDiameter = " + diameter() +
                      "\nCircumference = " + circumference() +
                      "\nArea = " + area() +
                      "\nVolume = " + volume());
  }  // end displayStatistics
} // end Sphere
```

The subclass *Ball* will inherit all the members of the class *Sphere*—except the constructor—and define additional methods and data fields. The *Ball* class could add:

A subclass can add new members to those it inherits

- A data field that names the ball
- Methods to access this name and set this name
- A method to alter an existing ball's radius and name
- A revised method *displayStatistics* to display the ball's name in addition to its statistics as a sphere

A subclass can override an inherited method of its superclass

You can add as many new members to a subclass as you like. Although you cannot revise a superclass's private data fields and should not reuse their names, you can **override** methods in the superclass. A method in a subclass overrides a method in the superclass if the two methods have the same declarations. Here, the class *Ball* overrides *displayStatistics*. Figure 9-2 illustrates the relationship between *Sphere* and *Ball*.

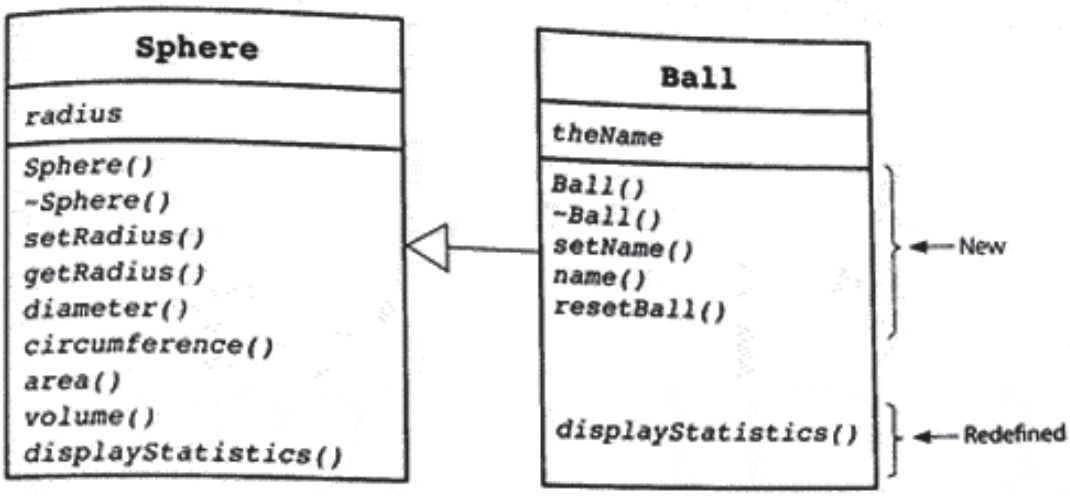


FIGURE 9-2

The subclass ***Ball*** inherits members of the superclass ***Sphere*** and overrides and adds methods

You can declare *Ball* as follows:

```
public class Ball extends Sphere {
  private String name;  // the ball's name

  // constructors:
  public Ball() {
  // Creates a ball with radius 1.0 and name "unknown."
    setName("unknown");
  }  // end default constructor

  public Ball(double initialRadius, String initialName) {
  // Creates a ball with radius initialRadius and
  // name initialName.
    super(initialRadius);
    setName(initialName);
  }  // end constructor

  // additional or revised operations:
  public boolean equals(Object rhs) {
    return ((rhs instanceof Ball) &&
            (getRadius() == ((Ball)rhs).getRadius()) &&
            (name.compareTo(((Ball)rhs).name)==0) );
  }  // end equals

  public String getName() {
  // Determines the name of a
```

This constructor calls **Sphere**'s default constructor implicitly

This constructor calls another ***Sphere*** constructor explicitly

```
    return name;
  }  // end getName

  public void setName(String newName) {
  // Sets (alters) the name of an existing ball.
    name = newName;
  }  // end setName

  public void resetBall(double newRadius, String newName) {
  // Sets (alters) the radius and name of an existing
  // ball to newRadius and newName, respectively.
    setRadius(newRadius);
    setName(newName);
  }  // end resetBall

  public void displayStatistics() {
  // Displays the statistics of a ball.
    System.out.print("\nStatistics for a "+ name());
    super.displayStatistics();
  }  // end displayStatistics

}  // end Ball
```

Adding *extends Sphere* after *class Ball* indicates that *Sphere* is a superclass of *Ball* or, equivalently, that *Ball* is a subclass of *Sphere*.

An instance of a subclass has all the behaviors of its superclass

An instance of *Ball* has two data fields—*radius*, which is inherited, and *name*, which is new. An instance of a subclass can invoke any public method in the superclass. Thus, an instance of *Ball* has all the methods that *Sphere* defines; new constructors; new methods *getName*, *setName*, and *resetBall*; and revised methods *equals* and *displayStatistics*.

A subclass inherits private members from the superclass, but cannot access them directly

A subclass cannot access the private members of the superclass directly, even though they are inherited. Inheritance does not imply access. After all, you can inherit a locked vault but not be able to open it. In the current example, the data field *radius* of *Sphere* is private, so you can reference it only within the definition of *Sphere* and not within the definition of *Ball*. However, *Ball* can use *Sphere*'s public methods *setRadius* and *radius* to set or obtain the value of *radius* indirectly, as is done in the revised *equals* method in the *Ball* class.

A subclass's methods can call the superclass's public methods

Within the implementation of *Ball*, you can use the methods that *Ball* inherits from *Sphere*. For example, the new method *resetBall* calls the inherited method *setRadius*. Also, *Ball*'s *displayStatistics* calls the inherited version of *displayStatistics*, which you indicate by writing *super.displayStatistics()*. The word *super* represents the object reference for the superclass and is necessary to differentiate between the two versions of the method. Thus, you can access a superclass method, even though it has been overridden, by using the *super* reference.

In the version of `displayStatistics` in the `Ball` class, the subclass overrides the superclass version of the method. The subclass version calls the superclass version of the method and performs additional tasks as well.

Java **annotations** provides a mechanism for a programmer to explicitly notify the compiler that a method from the superclass is being overridden. Annotations have a number of uses including providing additional information to the compiler to detect errors, informing the compiler to suppress warnings, and warning of the use of deprecated elements. Of interest here is the annotation `@Override`—it is used to indicate that the annotated method is overriding a method in a super class. If a method with this annotation does not override its superclass's method because you misspell the method name or do not correctly match the parameters, the compiler will generate an error. It also makes your code easier to read and understand because the fact that you are overriding the method is explicitly stated in the code.

The annotation `@Override` indicates that a method from the superclass is being overridden

To use the `@Override` annotation, simply place it on the line preceding the overriding method:

```
@Override
public void myMethod() {  }
```

So for example, in the `Ball` class, `displaystatistics` would now appear as follows:

```
@Override
public void displayStatistics() {
// Displays the statistics of a ball.
  System.out.print("\nStatistics for a "+ name());
  super.displayStatistics();
} // end displayStatistics
```

Clients of a subclass can invoke the public members of the superclass. For example, if you write

Clients of a subclass can invoke the superclass's public methods

```
Ball myBall = new Ball(5.0, "Volleyball");
```

`myBall.diameter()` returns `myBall`'s diameter, 10.0 (2 times `myBall`'s radius), by using the method `diameter` that `Ball` inherits from `Sphere`. If a new method has the same name as a superclass method—`displayStatistics`, for example—instances of the new class will use the new method, while instances of the superclass will use the original method. Therefore, if `mySphere` is an instance of `Sphere`, the call `mySphere.displayStatistics()` will invoke `Sphere`'s `displayStatistics`, whereas `myBall.displayStatistics()` will invoke `Ball`'s `displayStatistics`, as Figure 9-3 illustrates.

Finally, note that before any constructor in the subclass is executed, the default constructor (the constructor with no parameters) of the superclass will be executed unless you've specified an alternative constructor from the superclass. To call an alternative constructor, [illegible] with the arguments for the

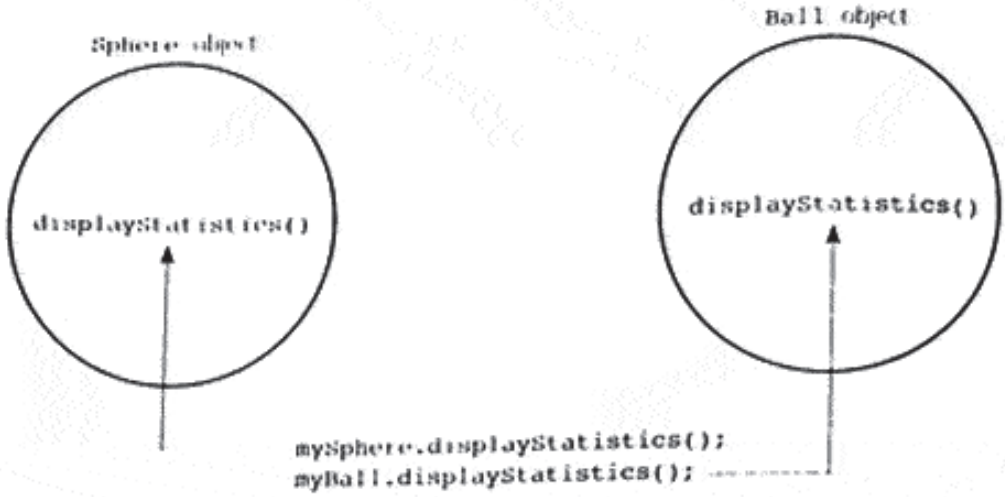


FIGURE 9-3
An object invokes the correct version of a method

superclass constructor you want to use. For example, the class `Ball` constructor `Ball(double initialRadius, String initialName)` calls the single-parameter constructor `Sphere(double initialRadius)` by using the statement `super(initialRadius)`. This call must appear as the first statement of the subclass's constructor.

Because the default constructor for the class `Ball` does not specify an alternative constructor, the `Sphere` constructor with no parameters is automatically called. Hence, if an instance of the class `Ball` is created using the following statement:

```
Ball myBall = new Ball();
```

`myBall` will have a radius of 1.0 and `"unknown"` as its name.

Java Access Modifiers

A class with no access modifier is available to other classes within the same package

The keywords `public` and `private` are called **access modifiers.** They are used to control the visibility of the members of a class. When no access modifier is specified on a member declaration, the member is visible only to classes in the same package. In addition to the `public` and `private` access modifiers, the `protected` access modifier allows a class designer to hide the members from a class's clients but make them available to a subclass or to another class within the same package. That is, a subclass can reference the protected members of its superclass directly, but clients of the superclass or subclass cannot. Also, classes within the same package can access protected members directly.

For example, `Sphere` has a private field `radius`, which the subclass `Ball` cannot reference directly. If, instead, you declared `radius` as protected, the class `Ball` would be able to access `radius` directly. Any other classes within the same package would also be able to access `radius` directly. Clients of `Ball` or `Sphere`, however, would not have direct access to `radius`. Figure 9-4

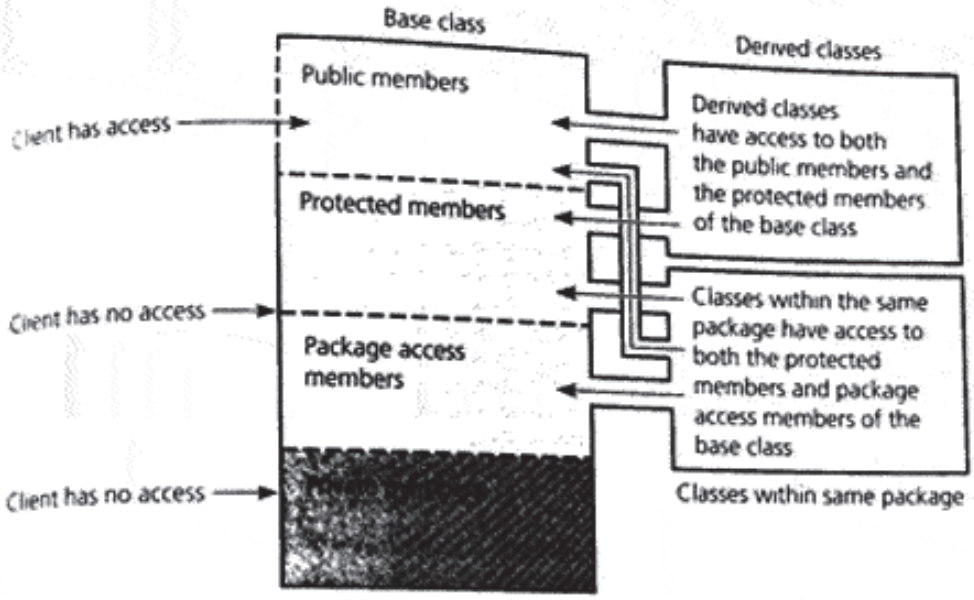


FIGURE 9-4
Access to public, protected, package access, and private members of a class by a client and a subclass

illustrates public, private, protected, and package (default) access for members of a class.

In general, a class's data fields should be private

As a general stylistic guideline, you should make all data fields of a public class private and, if desired, provide indirect access to them by defining methods that are either public or protected. Although a class's public members are available to anyone, its protected members are available exclusively to either its own methods, members of other classes in the same package, or the methods of a subclass. The following summary distinguishes among the access modifiers for members of a class:

KEY CONCEPTS

Membership Categories of a Class

1. Public members can be used by anyone.
2. Members declared without an access modifier (the default) are available to methods of the class and methods of other classes in the same package.
3. Private members can be used only by methods of the class.
4. Protected members can be used only by methods of the class, methods of other classes in the same package, and methods of the subclasses.

Is-a and Has-a Relationships

As you just saw, inheritance provides for superclass/subclass relationships among classes. Other relationships are also possible. When designing new classes from existing ones, it is important to identify their relationship so that you can determine whether to use inheritance. Two basic kinds of relationships are possible: *is-a* and *has-a*.

Inheritance should imply an is-a relationship

***Is-a* relationships.** Earlier in this chapter, we used inheritance to derive the class `Ball` from `Sphere`. You should use inheritance only when an *is-a* relationship exists between the superclass and the subclass. In this example, a ball *is a* sphere, as Figure 9-5 illustrates. That is, whatever is true of the superclass `Sphere` is also true of the subclass `Ball`. Wherever you can use an object of type `Sphere`, you can also use an object of type `Ball`. This feature is called **object type compatibility.** In general, a subclass is type-compatible with all of its superclasses. Thus, you can use an instance of a subclass instead of an instance of its superclass, but not the other way around.

You can use an instance of a subclass anywhere you can use an instance of the superclass

The object type of an actual argument in a call to a method can be a subclass of the object type of the corresponding formal parameter. As you've seen in many of the implementations of ADTs presented earlier, you can use the class `Object` as the type of the formal parameter, and since all classes are derived from the `Object` class, you can use any object type as the actual argument.

As another example, suppose your program uses `Sphere` and `Ball` and contains the following static method:

```
public static void displayDiameter(Sphere aSphere) {
   System.out.println("The diameter is "
                      + aSphere.diameter() + ".");
}  // end displayDiameter
```

FIGURE 9-5
A ball "is a" sphere

If you define *mySphere* and *myBall* as

```
Sphere mySphere = new Sphere(2.0);
Ball myBall = new Ball(5.0, "Volleyball");
```

the following calls to *displayDiameter* are legal:

```
displayDiameter(mySphere);   // mySphere's diameter
displayDiameter(myBall);     // myBall's diameter
```

The first call is unremarkable because both the actual argument *mySphere* and the formal parameter *aSphere* have the same data type. The second call is more interesting: The type of the actual argument *myBall* is a subclass of the data type of the formal parameter *aSphere*. Because a ball is a sphere, it can behave like a sphere. That is, *myBall* can perform sphere behaviors, so you can use *myBall* anywhere you can use *mySphere*.

Since a ball is a sphere, you can use it anywhere you can use a sphere

***Has-a* relationships.** A ball-point pen *has a* ball as its point, as Figure 9-6 illustrates. Although you would want to use *Ball* in your definition of a class *Pen*, you should not use inheritance, because a pen is not a ball. In fact, you do not use inheritance at all to implement a *has-a* relationship. Instead, you can define a data field *point*—whose type is *Ball*—within the class *Pen*, as follows:

If the relationship between two classes is not is-a, you should not use inheritance

```
public class Pen {
   private Ball point;
   ...
}  // end Pen
```

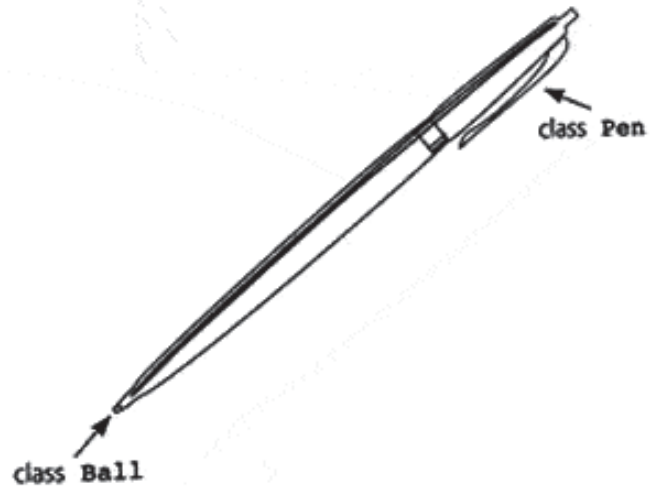


FIGURE 9-6
A pen "has a" or "contains a" ball.

An instance of `Pen` has, or *contains*, an instance of `Ball`. Thus, another name for the *has-a* relationship is **containment**.

You have already seen two other examples of the *has-a* relationship among classes in the preceding two chapters: Chapter 7 presented an implementation of `StackInterface` that used the ADT list to represent the items in a stack, while Chapter 8 used a similar implementation for the ADT queue. The class `StackListBased`, for example, contains a private data field `list` of type `ListReferenceBased`. That is, an instance of `StackListBased` has, or contains, an instance of `ListReferenceBased` that manages the stack's items.

The *has-a* relationship between two classes is possible when inheritance is inappropriate. Later, this chapter implements the ADT sorted list by using the two relationships just discussed.

9.2 Dynamic Binding and Abstract Classes

As you saw earlier, if `mySphere` is an instance of `Sphere` and `myBall` is an instance of `Ball`, `mySphere.displayStatistics()` invokes `Sphere`'s version of `displayStatistics`, whereas `myBall.displayStatistics()` invokes `Ball`'s version of `displayStatistics`. (See Figure 9-3.) Suppose, however, that the following statements are executed:

```
Ball myBall = new Ball(1.25, "golfball");
Sphere mySphere = myBall;
mySphere.displayStatistics();
```

Since `mySphere` actually references an instance of `Ball`, the `Ball` version of `displayStatistics` is executed. Thus, the appropriate version of a method is decided at execution time, instead of at compilation time, based on the type of object referenced. This situation is called **late binding**, or **dynamic binding**, and a method such as `displayStatistics` is called **polymorphic**. We also say that `Ball`'s version of `displayStatistics` **overrides** `Sphere`'s version.

A polymorphic method has multiple meanings

We will now examine a more subtle example of late binding. Suppose you wanted the class `Ball` to have a method `area` that behaved differently than `Sphere`'s `area`. Just as `Ball` overrides `displayStatistics`, it could override `area` to compute, for example, the ball's cross-sectional area, which is used to compute the drag on a ball. Thus, you would add the method

```
@Override
public double area() {  // cross-sectional area
   double r = getRadius()
   return Math.PI * r * r;  // Math.PI is a constant
}  // end area
```

is a public member of the class *Ball*, overriding *Sphere*'s *area*. Consider

```
public class Sphere {
  . . .     // everything as before
  public double area() {  // surface area
    return 4.0 * Math.PI * radius * radius;
  }  // end area

  public void displayStatistics() {
    System.out.println("\nRadius = " + getRadius()
                + "\nDiameter = " + diameter()
                + "\nCircumference = " + circumference()
                + "\nArea = " + area()
                + "\nVolume = " + volume());
  }  // end displayStatistics

  . . .
}  // end Sphere

public class Ball extends Sphere {
  . . .  // everything as before, except
         // displayStatistics is omitted
         // and area is revised:
  @Override
  public double area() {
    double r = getRadius();
    return Math.PI * r * r;
  }  // end area
  . . .
}  // end Ball
```

Now when an instance of *Sphere* calls *displayStatistics*, *displayStatistics* will call *Sphere*'s *area* (Figure 9-7a), yet when an instance of *Ball* calls *displayStatistics*, *displayStatistics* will call *Ball*'s *area* (Figure 9-7b). Thus, the meaning of *displayStatistics* depends on the type of object that invokes it.

You can control whether a subclass can override a superclass method

The designer of a superclass does have some control over whether a subclass is allowed to override a superclass method. If the field modifier *final* is specified in the method definition (typically after other field modifiers such as *public* and *static*), the method cannot be overridden by a subclass. On the other hand, the field modifier *abstract* requires the subclass to override the method. Abstract methods are discussed in more detail in the next section.

Methods that are **final** or **static** use static binding

When a method is defined as *final*, the compiler can determine which form of a method to use at compilation time—as opposed to at execution time. This situation is called **early binding, or static binding.** Methods declared *static* also use static binding, since only one version of the method is available for all classes.

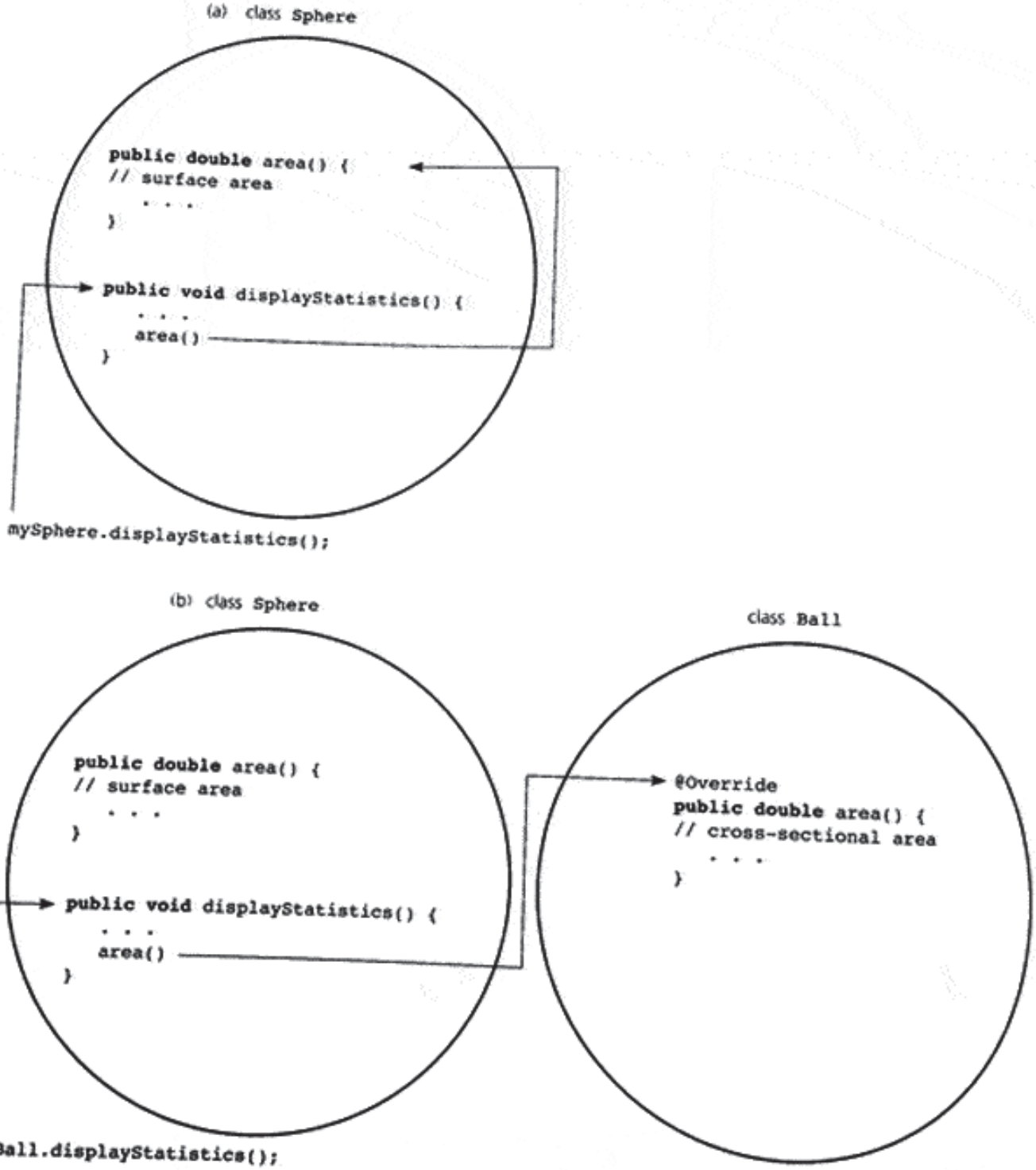


FIGURE 9-7

area is overridden: (a) ***mySphere.displayStatistics()*** calls ***area*** in ***Sphere***; (b) ***myBall.displayStatistics()*** calls ***area*** in ***Ball***

Overloading methods. When you override a method, you create a method that has the same name and same set of parameters as the original method. Sometimes it is convenient to define another method with the same name as the first but with a different set of parameters. Such a method **overloads** the first method. Frequently you overload constructors so that each constructor has a different set of parameters. Often the choice of which constructor to use depends on the information available when you create an instance. Some of the constructors require fewer parameters and will set some of the data fields to default values.

For another example of overloading, consider the class *Ball* and the method *resetBall*, defined as follows:

```
public void resetBall(double newRadius, String newName) {
   Sets (alters) the radius and name of an existing
   ball to newRadius and newName, respectively.
  setRadius(newRadius);
  setName(newName);
}  // end resetBall
```

You could define two other methods to reset the data fields:

```
public void resetBall(double newRadius) {
// Sets (alters) the radius of an existing
// ball to newRadius.
  setRadius(newRadius);
}  // end resetBall

public void resetBall(String newName) {
// Sets (alters) the name of an existing
// ball to newName.
  setName(newName);
}  // end resetBall
```

All three methods have the same name but different sets of parameters. The arguments in each call to *resetBall* determine which version of the method will be used.

Abstract Classes

Suppose you have a CD player (CDRW) and a DVD player (DVDRW). Both devices share several characteristics. Each involve an optical disc. You can insert, remove, play, record, and stop such discs. Some of these operations are essentially the same for both devices, while others—in particular, the play and record methods—are different but similar.

If you were specifying both devices, you might begin by describing the common operations:

Disc transport operations

```
+insert()
// Inserts a disc into the player.

+remove()
// Removes a disc from the player.

+play()
// Plays the disc.

+record()
// Record the disc.

+stop()
// Stops playing the disc.

+skipForward()
// Skip ahead to another section of the disc.

+skipBackward()
// Skip back to an earlier section of the disc.
```

These operations could constitute a generic disc player (GDRW).

If GDRW, CDRW, and DVDRW were classes, GDRW could be the base class of CDRW and DVDRW, as Figure 9-8 illustrates. While GDRW could implement operations such as *insert* and *remove* that would be suitable for both a CDRW and a DVDRW, it could only indicate that these devices have a *play* and *record* operation. So CDRW, for example, inherits the operations

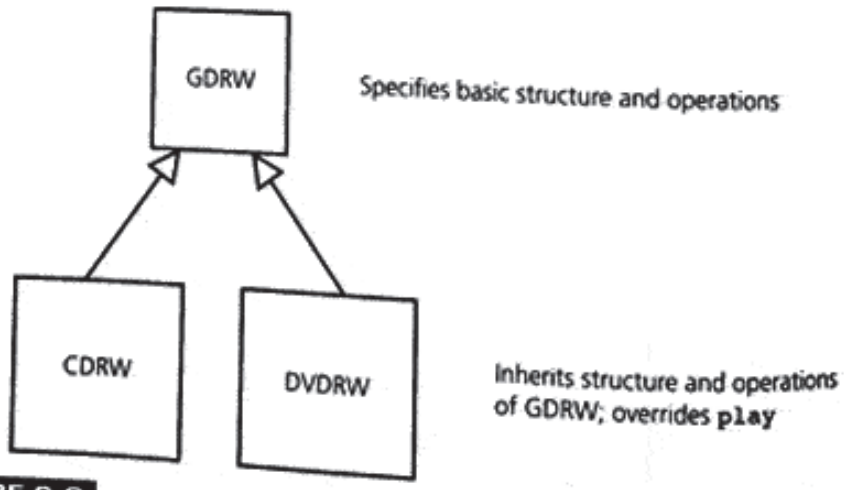


FIGURE 9-8
CDRW and DVDRW have an abstract base class GDRW

provided by GDRW but overrides the `play` and `record` operation to suit CDs, as Figure 9-9 illustrates. If necessary, CDRW could override any of GDRW's operations or define additional ones. We can make similar comments about DVDRW. Thus,

- A CDRW is a GDRW that plays sound.
- A DVDRW is a GDRW that plays sound and video.

Because GDRW cannot implement its play or record operations, we would not want instances of it. So GDRW is simply a class without instances that forms the basis of other classes. If a class never has instances, its methods need not be implemented. Such methods, however, must be abstract so that subclasses can supply their own implementations.

An **abstract class** has no instances and is used only as the basis of other classes. Thus, the general disc player is an abstract class. In Java, you declare an abstract class by including the keyword `abstract` in the class definition.

A class that contains at least one abstract method is an abstract class

An abstract class, like other classes, can contain both data fields and methods. Although an abstract class contains the methods and data fields common to all of its subclasses, some of its methods might have their implementations deferred to the subclasses. In these cases, you also declare the methods themselves to be abstract by including the field modifier `abstract` in the method definition. An abstract method does not have a body; instead, the method heading ends with a semicolon. For example, the following is a declaration for an abstract method:

```
public abstract void record();
```

Any class that contains at least one abstract method must itself be declared abstract. Any subclass that fails to implement all of the abstract methods in its superclass must also be declared as an abstract class.

An abstract class has subclasses but no instances

Another example. The previous classes `Sphere` and `Ball` describe points that are equidistant from the origin of a three-dimensional coordinate system. The

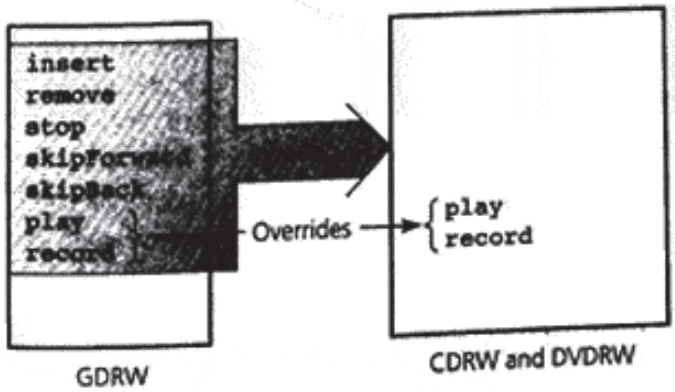


FIGURE 9-9
CDRW and DVDRW are subclasses of GDRW

following class, *EquidistantShape*, which declares operations to set and return the distance of a point from the origin, could be an abstract class of *Sphere*:

An abstract class of **Sphere**

```
public abstract class EquidistantShape {
  private double radius;
  public static final double DEFAULT_RADIUS = 1.0;

  public void setRadius(double newRadius) {
    if (newRadius >= 0.0) {
      radius = newRadius;
    }  // end if
  }  // end setRadius

  public double getRadius() {
    return radius;
  }  // end getRadius

  public abstract double area();
  public abstract void displayStatistics();
}  // end EquidistantShape
```

This class declares a private data field *radius* with the public methods *setRadius* and *getRadius* to access *radius*. The class also contains two abstract methods, *area* and *displayStatistics*.

Now you could define the class *Sphere* as a subclass of *EquidistantShape*:

```
public class Sphere extends EquidistantShape {
  public Sphere() {
    setRadius(1.0);
  } // end default constructor

  public Sphere(double initialRadius) {
    setRadius(initialRadius);
  } // end constructor

  // Implementation of abstract methods

  @Override
  public double area() {
    double r = getRadius();
    return 4.0 * Math.PI * r * r;
  } // end area

  @Override
  public void displayStatistics...
```

```
public void displayStatistics() {
  System.out.println("\nRadius = " + getRadius()
                     + "\nDiameter = " + diameter()
                     + "\nCircumference = " + circumference()
                     + "\nArea = " + area()
                     + "\nVolume = " + volume());
    end displayStatistics

  Remaining methods for class appear here
. . .

  end Sphere
```

By including the abstract method **displayStatistics** in the abstract class EquidistantShape, you force some subclass—like **Sphere**—to implement it. The class Ball can now be a subclass of **Sphere**, just as it appeared earlier in the chapter.

Note that radius is a private data field of **EquidistantShape** instead of Sphere. This means that subclasses of **Sphere** will be unable to access **radius** directly by name. Thus, the class **EquidistantShape** must contain the methods setRadius and getRadius, and you must provide default implementations for the subclasses to inherit. Alternatively, you could define **radius** to be a protected data field of **EquidistantShape**, enabling the subclass to both access radius directly and define **setRadius** and **getRadius**. Although data fields are generally private, a protected data field within an abstract class is reasonable, since the class is always a superclass of another class. That is, an abstract class has another purpose but to form the basis of another class.

An abstract class can provide a constructor, but the constructors cannot be abstract. The key points about abstract classes are summarized as follows:

KEY CONCEPTS

Abstract Classes

1. An abstract class is used only as the basis for subclasses and thus defines a minimum set of methods and data fields for its subclasses.
2. An abstract class has no instances.
3. An abstract class should, in general, omit implementations except for the methods that provide access to private data fields or that express functionality common to all of the subclasses.
4. A class that contains at least one abstract method must be declared as an abstract class.
5. A subclass of an abstract class must be declared abstract if it does not provide implementations for all abstract methods in the superclass.

Java Interfaces Revisited

A Java interface specifies the common behavior of a set of classes

Inheritance is one way to have a common set of methods (or common behavior) available for a group of classes. Inheritance also allows the subclasses to inherit the structure of the superclass. In some instances, however, it is only the behavior that is of interest. Java interfaces provide another mechanism for specifying common behavior for a set of (perhaps unrelated) classes.

You have already seen how to use Java interfaces to specify the methods for a class. The methods were then implemented by using a variety of techniques, including array-based and reference-based implementations. Clients of the class could use one of these implementations. To facilitate moving from one implementation to another, you can declare references to the class by using the interface definition instead of the class definition. For example, the class `StackArrayBased` implements `StackInterface`. A client of the ADT stack could contain the following statement:

A client can reference a class's interface instead of the class itself

```
StackInterface stack = new StackArrayBased();
```

Since `stack` has the type `StackInterface`, you can use only methods that appear in the interface definition with `stack`—for example, `stack.pop()`. The interface definition should also be used whenever the stack appears as a formal parameter in a method. If you do this throughout the client, all you will need to do to move to the reference-based implementation will be to change the places where instances of the stack are created. For example, the previous definition of `stack` would change to

```
StackInterface stack = new StackReferenceBased();
```

and method calls that adhere to the interface, such as `stack.pop()`, would continue to work in the client.

An interface specifies behaviors that are common to a group of classes

As was mentioned earlier, another common use of interfaces is to specify behavior that is common to a group of unrelated classes. For example, you could define an interface called `AnimateInterface`, as follows:

```
public interface AnimateInterface {
  public void move(int x, int y);
  public void paint();
}  // end AnimateInterface
```

The intent is to provide a common set of methods needed for screen animation. Thus, many different classes, such as the class `Ball`, could implement this interface so that instances of `Ball` could be animated on the screen. If the class `Ball` does not implement all of the methods in the interface, with the intent that a subclass will implement the missing methods, the `Ball` class must be declared as abstract.

You can use inheritance to define a subinterface

As you saw in the discussion of the Java Collections Framework in Chapter 5, you can use inheritance to derive new interfaces, often called **subinterfaces**. In particular, the basis for the ADT collections in the JCF is the interface

`java.util.Iterable`, with the subinterface `java.util.Collection`. This allowed the `Collection` interface to inherit a method called `Iterator` that returns an `Iterator` object for the collection. Later in the chapter, we will examine how to create an iterator for our own collection classes.

9.3 Java Generics

Generic Classes

The ADTs developed in this text thus far relied upon the use of the `Object` class as the data type for the elements. For example, the interface `ListInterface` in the previous section used `Object` as the data type for the list items. Because of polymorphism, we could use objects of any class as items in the list. But this approach has some issues:

- Though the ADTs were intended to be used as homogeneous data structures, in reality, items of any type could be added to the same ADT instance.
- To use objects returned from the ADT instance, we usually had to cast the object back to the actual type in use.

The second issue may lead to class-cast exceptions if the items removed from the ADT instance are not of the type expected.

A generic class describes a class in terms of a data-type parameter

You can avoid these issues by using Java generics to specify a class in terms of a data-type parameter. When you (the client) declare instances of the class, you specify the actual data type that the parameter represents. We have seen such declarations in our discussions of the Java Collections Framework.

Here is a simple generic class definition, where `T` is the formal data-type parameter:

```
public class NewClass <T> {

  private int year;
  private T data = null;

  public NewClass() {
    year = 1970;
  }  // end constructor

  public NewClass(T initialData) {
    year = 1970;
    data = initialData;
  }  // end constructor

  public NewClass(T initialData, int year) {
    this.year = year;
    data = initialData;
  }  // end constructor
```

```
  public void setData(T newData) {
    data = newData;
  }  // end setData

  public T getData() {
    return data;
  }  // end getData

  public String toString() {
    if (data != null) {
      return data.toString() + ", " + year;
    }
    else {
      return null + ", " + year;
    }  // end if
  }  // end toString
}  // end NewClass
```

You follow the class definition with the data-type parameter enclosed in < >. If there is more than one type to be parameterized, they are separated by commas. In the implementation of the class, you use the data-type parameter exactly as you would any other type.

A simple program that uses this generic class could begin as follows:

The client specifies an actual data type when declaring an instance of the class

```
static public void main(String[] args) {
  NewClass<String> first = new NewClass<String>("Wally", 2010);
  NewClass<Integer> second = new NewClass<Integer>(15);

  System.out.println("Contents of first => " + first);
  first.setData("Wood");
  System.out.println("After modifying first => " + first);
  System.out.println("Result of getData on second=> " +
                                                  second.getData());

  ...
} // end main
```

Primitive types are not allowed as type-parameters

Notice that the declarations of *first* and *second* specify the data type that the parameter *T* represents within the generic class. When using a generic class, the data-type parameters should always be included. Primitive types are not allowed as generic type-parameters.

The Java compiler will allow generic classes without data type parameters to be declared, but it is primarily for backward compatibility with code written prior to generics being included in the language. In the absence of a data type being specified, the compiler will generate warning messages when actual instances are used where instances of the data-type parameter are expected.

You must be careful about what you do with objects of the data-type parameter within the implementation of a generic class. For example, note the `toString` method in `NewClass`. It utilizes the `toString` method of `data` that has been declared of type `T`. This will use the definition for `toString` that exists for `T`, so if the `toString` method for that type does not override the one provided by the class `Object`, you will get the default string representation provided by the class `Object`, which includes the class name and hash code for the object.

Finally, Java does not allow generic types to be used in array declarations. When you declare an array with a generic type and attempt to instantiate it, you will get the following error message:

```
Error: generic array creation
  T[] test = new T[10];
```

The alternative is to use either the `ArrayList` or `Vector` class in the Java Collections Framework using the data-type parameter as follows:

```
Vector<T> test = new Vector<T>();
ArrayList<T> test2 = new ArrayList<T>();
```

Generic Wildcards

Instances of generic classes are not related

Note that when generic classes are instantiated, they are not necessarily related. For example, if we try to assign `second` to `first` in the above code, we get the following error message:

```
Error: incompatible types
     found   : NewClass<java.lang.Integer>
     required: NewClass<java.lang.String>
       first = second;
```

The instances `first` and `second` are considered to be of two different types. But there are situations where it would be convenient to write code that could handle both of these instances based upon the fact that they utilize the same generic class. This can be indicated by using the `?` wildcard as the data-type parameter, where the `?` stands for an unknown type. For example, the method:

```
public void process(NewClass<?> temp) {
  System.out.println("getData() => " + temp.getData());
}  // end process
```

can be used to process both the `first` and `second` instances.

Generic Classes and Inheritance

Generic classes can be used with inheritance

You can still use inheritance with a generic class or interface. You can specify actual data-types that should be used, or allow the subclass to maintain the same data-type parameter by utilizing the same name in the declaration. Additional data-type parameters may also be specified. For example, given the generic class *Book* defined as follows:

```
public class Book<T, S, R>
```

The following are legal subclasses of *Book*:

```
// Uses same generic parameters
public class RuleBook<T, S, R> extends Book<T, S, R>

// Specifies actual types for all of the type parameters
public class MyBook extends Book<Integer, String, String>

// Specifies the types for some of the type parameters and adds an
// additional one Q
public class TextBook<T, Q> extends Book<T, String, String>
```

Note that the rules of method overriding are in effect, a method (with the same name) defined in a subclass will override a method in the superclass if:

- you declare a method with the same parameters in the subclass, and
- the return type of the method is a subtype of all the methods it overrides.

The second point was introduced in Java 1.5.

To further our discussion of some of the other features of Java generics, assume that we have the class hierarchy shown in Figure 9-10. Note that the class *Object* is at the root of the hierarchy, *Person* is a subclass of *Object*, *Student* is a subclass of *Person*, and *UgradStudent* and *GradStudent* are subclasses of *Student*.

When specifying a generic class, it is sometimes useful to indicate that the data-type parameter should be constrained to a class or one of its subclasses or an implementation of a particular interface. To indicate this, you use the keyword *extends* to indicate that the type should be a subclass or an implementation of an interface. The following definition of the interface *Registration* restricts the generic parameter to *Student* or classes that extend *Student*:

```
public interface Registration <T extends Student> {

  public void register(T student, CourseID cid);
  public void drop(T student, CourseID cid);
  ...
}  // end Registration
```

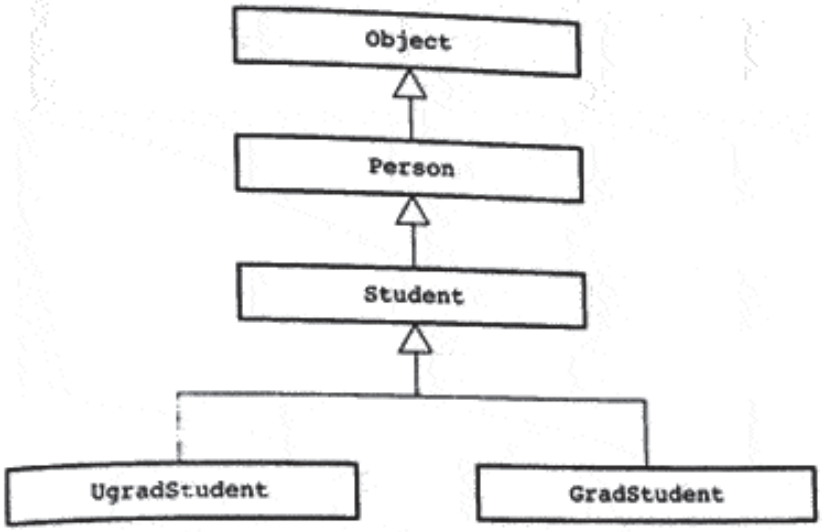


FIGURE 9-10
Sample class hierarchy

So the following declarations would be allowed:

```
Registration<Student> students = new Registration<Student>();
Registration<UpgradStudent> ugrads = new Registration<UgradStudent>();
Registration<GradStudent> grads = new Registration<GradStudent>();
```

Attempting to use a class that is not a subclass of *Student* will result in a compile-time error. For example:

```
Registration<Person> people = new Registration<Person>();
```

generates an error similar to this:

```
Error: type parameter Person is not within its bound
```

Hence, use of the *extends* clause is a way of constraining or placing an *upper bound* on the data-type parameter.

The **extends** clause places an *upper bound* on the data-type parameter

The *extends* clause can also be used to bound the ? wildcard discussed earlier. For example, the following declaration could process any *ArrayList*:

```
Public void process(ArrayList<?> list)
```

But you might want the method to be constrained to a list containing objects of type *Student* or one of its subclasses as follows:

```
Public void process(ArrayList<? extends Student> stuList)
```

In this case, a call to the method *process* might look like this:

```
ArrayList<UgradStudent> ugList =
                                        new ArrayList<UgradStudent>();
test1.process(ugList);
```

But if an attempt (similar to the following) to use a class that is not of type *Student* or one of its subclasses is made,

```
ArrayList<Person>pList = new ArrayList<Person>();
test1.process(pList);
```

an error message similar to the following will be generated by the compiler:

```
Error: process(java.util.ArrayList<? extends Student>) in Test
cannot be applied to (java.util.ArrayList<Person>)
```

Sometimes, the specification of a type can be too restrictive. One common scenario where this occurs is when the *Comparable* interface is involved. Let's assume that the *Student* class extends the *Comparable* interface as follows:

```
class Student extends Person implements Comparable<Student> {
  protected String id;
  ...

  public int compareTo(Student s) {
    return id.compareTo(s.id);
  } // end compareTo
  ...
} // end Student
```

Furthermore, assume that we have a class defined as follows:

```
import java.util.ArrayList;
class MyList <T extends Comparable<T>> {

  ArrayList<T> list = new ArrayList<T>();

  public void add(T x) {
    ...
    if ((list.get(i)).compareTo(list.get(j)) < 0) {
      ...
    } // end if
  } // end add
  ...
} // end MyList
```

Note that the following declaration compiles:

```
MyList<Student> it320 = new MyList<Student>();
```

But that the declaration:

```
MyList<UgradStudent> it321 = new MyList<UgradStudent>();
```

produces an error message similar to the following:

```
Error: type parameter UgradStudent is not within its bound
```

Note that the data-type parameter for *MyList* expects a class that implements the *Comparable* interface for *T*. The class *UgradStudent* does not implement this directly, it is inherited from the superclass *Student*. To allow the *UgradStudent* class as the parameter to the generic class *MyList*, the class must define the data-type parameter to allow for a superclass of *T* to implement the *Comparable* interface. This is done as follows:

```
class MyList <T extends Comparable<? super T>>
```

The clause *<? super T>* specifies a *lower bound* on the data-type parameter. In essence, it is a way to say that the class or one of its superclasses can be used as the actual data-type parameter.

The **super** clause places a *lower bound* on the data-type parameter

Generic Implementation of the Class List

The following files revise the reference-based list class—which appears in Chapter 5 and was discussed again earlier in this chapter—as a generic class. Differences between this generic version and the earlier version are shaded. The data-type parameter *T* is used instead of *Object*.

```
// ****************************************************
// Interface for the ADT list
// ****************************************************
public interface ListInterface<T> {

  public int size();
  public boolean isEmpty();
  public void removeAll();
  public void add(int index, T item)
                  throws ListIndexOutOfBoundsException;
  public void remove(int index)
                  throws ListIndexOutOfBoundsException;
  public T get(int index)
                  throws ListIndexOutOfBoundsException;
} // end ListInterface
```

```
// ******************************************************
// Class Node used in the implementation of ADT list
// ******************************************************
class Node <T> {
  T item;
  Node<T> next;

  public Node(T newItem) {
    item = newItem;
    next = null;
  } // end constructor

  public Node(T newItem, Node<T> nextNode) {
    item = newItem;
    next = nextNode;
  } // end constructor

} // end class Node

// ******************************************************
// Reference-based implementation of ADT list.
// ******************************************************
public class ListReferenceBased<T> implements ListInterface [redacted] {
  // reference to linked list of items
  private Node<T> head;
  private int numItems; // number of items in list

  public ListReferenceBased() {
    numItems = 0;
    head = null;
  }  // end default constructor

  public boolean isEmpty() {
    return numItems == 0;
  }  // end isEmpty

  public int size() {
    return numItems;
  }  // end size

  private Node[redacted] find(int index) {
    Node[redacted] curr = head;
    for (int skip = 1; skip < index; skip++) {
      curr = curr.next;
    } // end for
    return curr;
  } // end find
```

```
  // The methods get, add, and remove are omitted here – see
  // Exercise 12.

  public void removeAll() {
    // setting head to null causes list to be
    // unreachable and thus marked for garbage
    // collection
    head = null;
    numItems = 0;
  } // end removeAll
} // end ListReferenceBased
```

Generic Methods

Just like class and interface declarations, method declarations can also be generic. Methods, both static and non-static, and constructors can have data-type parameters. Like classes and interfaces, the declaration of the formal data-type parameters appears within the < > brackets, immediately before the method's return type. For example, here is a method to sort an *ArrayList*, where the elements in the *ArrayList* must belong to a class that implements the *Comparable* interface:

```
class MyMethods {

  public static <T extends Comparable<? super T>>
         void sort(ArrayList<T> list) {
    // implementation of sort appears here
  } // end sort
} // end MyMethods
```

To invoke a generic method, you simply call the method just as you would a non-generic method. Generic methods are invoked like regular non-generic methods. The user doesn't need to explicitly specify the actual data-type parameters; the compiler automatically determines this by using the actual arguments provided in the method invocation. For example:

```
class TestMethod {
  public static void main (String[ ] args) {
    ArrayList<String> names = new ArrayList<String>();
    names.add("Janet");
    names.add("Andrew");
    names.add("Sarah");
    . . .
    MyMethods.sort(names);
  } // end main
} // end TestMethod
```

Since the names in the list are of type *String*, the compiler automatically determines that the actual data-type argument for *T* is *String*.

9.4 The ADTs List and Sorted List Revisited

Chapter 4 introduced the ADT list and the ADT sorted list. As you know, these lists have some characteristics and operations in common. For example, each ADT can determine its length, determine whether it is empty, and remove all items from the list. Both ADTs also provide a retrieval operation that returns an object at a given index position in the list. You can organize such commonalities into an interface, which can be the basis of these and other list operations. For example,

An interface for lists and sorted lists

```
public interface BasicADTInterface {
  public int size();
  public boolean isEmpty();
  public void removeAll();
}  // end BasicADTInterface
```

The designer of this interface wants all implementing classes to have the operations *size*, *isEmpty*, and *removeAll*.

Notice that the *BasicADTInterface* could be used as the interface for the three ADTs that we have studied thus far: list, stack, and queue. All of the ADTs had the methods *isEmpty* and *removeAll*, and they could easily have had the method *size* as well. Using *BasicADTInterface* would be a way to guarantee that all subinterfaces minimally will have these three methods. For example, here is a new interface definition based upon *BasicADTInterface* for the ADT list presented in section 9.3:

An interface for a list

```
public interface ListInterface<T> extends BasicADTInterface {
  public void add(int index, T item)
         throws ListIndexOutOfBoundsException;
  public T get(int index)
         throws ListIndexOutOfBoundsException;
  public void remove(int index)
         throws ListIndexOutOfBoundsException;
}  // end ListInterface
```

The implementation of *ListReferenceBased<T>*, started in section 9.3, is consistent with this interface, so the implementation is omitted here.

Implementations of the ADT Sorted List That Use the ADT List

Now suppose you want to define and implement a class for the ADT sorted list, whose operations are

ADT sorted list operations

```
createSortedList()
isEmpty():boolean {query}
size():integer {query}
sortedAdd(in newItem:ListItemType) throw ListException
sortedRemove(in anItem:ListItemType) throw ListException
removeAll()
get(in index:integer) throw ListIndexOutOfBoundsException
locateIndex(in anItem:ListItemType):integer {query}
```

The method `createSortedList` is implemented as a constructor for the class. The methods `isEmpty`, `size`, and `removeAll` have already been specified in the interface `BasicADTInterface`. The interface definition for the ADT sorted list extends `BasicADTInterface` to include the other methods. The elements of the sorted list have one additional requirement: They must implement the `Comparable` interface, as discussed in Chapter 5, so that the sorted list can order the elements. Thus, we have

```
public interface
      SortedListInterface<T extends Comparable<? super T>>
      extends BasicADTInterface {
  public void sortedAdd(T newItem) throws ListException;
  public T get (int index)
       throws ListIndexOutOfBoundsException;
  public int locateIndex(T anItem);
  public void sortedRemove(T anItem) throws ListException;
} // end SortedListInterface
```

You could, of course, use an array or a linked list to implement a sorted list, but such an approach would force you to repeat much of the corresponding implementations of `ListArrayBased` and `ListReferenceBased`. Fortunately, you can avoid this repetition by using one of the previously defined classes, `ListReferenceBased<T>`, to implement the class `SortedList`.

You can reuse **`ListArray-Based`** to implement **`SortedList`**

Two approaches are possible by using the *is-a* and *has-a* relationships between the new class `SortedList` and the existing class `ListReferenceBased`. In most cases, one of the approaches will be best. However, we will use the sorted list to demonstrate both approaches.

A sorted list *is a* list. Chapter 4 stated that the ADT list is simply a list of items that you reference by position number. If you maintained those items in sorted order, would you have a sorted list? Ignoring name differences, most operations for the ADT list *are* the same as the corresponding operations for

the ADT sorted list. The insertion and deletion operations differ, however, and the ADT sorted list has an additional operation, *locateIndex*.

You can insert an item into a sorted list by first using *locateIndex* to determine the position in the sorted list where the new item belongs. You then use *ListReferenceBased*'s *add* method to insert the item into that position in the list. You use a similar approach to delete an item from a sorted list.

An is-a relationship implies substitution

Thus, a sorted list *is a* list, so you can use inheritance. That is, the class *SortedList* can be a subclass of the class *ListReferenceBased*, inheriting *ListReferenceBased*'s members and implementing the additional methods specified in the interface *SortedListInterface*. Thus, we have the following:

```
public class SortedList<T extends Comparable<? super T>>
       extends ListReferenceBased<T>
       implements SortedListInterface<T> {

  public SortedList() {
    // invokes default constructor of superclass
  }  // end default constructor

  public void sortedAdd(T newItem) {
  // Adds an item to the list.
  // Precondition: None.
  // Postcondition: The item is added to the list in
  // sorted order.
    int newPosition = locateIndex(newItem);
    super.add(newPosition, newItem);
  }  // end sortedAdd

  public void sortedRemove(T anItem) throws ListException {
  // Removes an item from the list.
  // Precondition: None.
  // Postcondition: The item is removed from the list
  // and the sorted order maintained.
    int position = locateIndex(anItem);
    if ((anItem.compareTo(get(position))==0)) {
      super.remove(position);
    }
    else {
      throw new ListException("Sorted remove failed");
    }  // end if
  }  // end sortedRemove

  public int locateIndex(T anItem) {
  // Finds an item in the list.
  // Precondition: None.
  // Postcondition: If the item is in the list, its
  // index position in the list is returned.  If the
```

```
    ;tem is not in the list, the index of where it
    belongs in the list is returned.
   int index = 0;
     Loop invariant: anItem belongs after all the
     elements up to the element referenced by index
   while ( (index < size()) &&
           (anItem.compareTo(get(index)) > 0 ) ) {
     ++index;
   } // end while
   return index;
 } // end locateIndex
end SortedList
```

Note that by carefully designing the method implementations, especially `locateIndex`, we do not need access to any of the private data fields of the class `ListReferenceBased`. This means that we could just as easily have used a generic version of `ListArrayBased` as the superclass.

The class `SortedList` now has operations such as `isEmpty`, `size`, and `get`—which it inherits from `ListReferenceBased`—and `sortedAdd` and `sortedRemove`. Note also, however, that `SortedList` has also inherited the `add` and `remove` methods from `ListReferenceBased`. The availability of all of the methods from the superclass may or may not be desirable. In this case, the `add` method could potentially destroy the sorted list by making inappropriate insertions into the sorted list. In general, there are two techniques that you can use to solve this problem:

1. You can override the unwanted method with one that provides the correct semantics.
2. You can override the unwanted method with one that simply raises an exception indicating that the method is not supported.

In the present situation, the first technique does not provide a satisfactory solution. The `add` method in the superclass `ListReferenceBased` has a parameter that designates the position of the insertion, whereas the `add` method in the subclass `SortedList` does not. Even if we change the name `sortedAdd` to `add`, the subclass's `add` could not override the superclass's `add`, because the two `add` methods have different sets of parameters.

The second technique, however, does provide a solution. The subclass overrides the superclass's `add` method with a method that throws an `UnsupportedOperationException`. For example, the subclass could contain the following method:

```
public void add(int index, T item)
       throws UnsupportedOperationException {
  throw new UnsupportedOperationException();
}
```

When the *add* method is overridden in this way, the exception *UnsupportedOperationException* is thrown if an instance of *SortedList* invokes *add*.

A sorted list *has a* list as a member. If you do not have an *is-a* relationship between your new class and an existing class, inheritance is inappropriate. You may, however, be able to use an instance of the existing class to implement the new class. The following declaration of the class *SortedList* *has a* private data field that is an instance of *ListInterface* and that contains the items in the sorted list:

An instance of **ListInterface** can implement the sorted list

```
public class SortedList<T extends Comparable<? super T>>
       implements SortedListInterface<T> {
  private ListInterface<T> aList;

// constructors:
  public sortedList() {
    aList = new ListReferenceBased<T>();
  }  // end default constructor

// sorted list operations:
  public boolean isEmpty() {
    // To be implemented in Programming Problem 1
  }  // end isEmpty

  public int size() {
    // To be implemented in Programming Problem 1
  }  // end size

  public void sortedAdd(T newItem) {
    int newPosition = locateIndex(newItem);
    aList.add(newPosition, newItem);
  }  // end sortedAdd

  public void sortedRemove(T anItem) {
    // To be implemented in Programming Problem 1
  }  // end sortedRemove

  public T get(int position) {
    // To be implemented in Programming Problem 1
  }  // end get

  public int locateIndex(T anItem) {
    // To be implemented in Programming Problem 1
  }  // end locateIndex
```

```
  public void removeAll() {
    // to be implemented in Programming Problem 1
  } // end removeAll
} // end SortedList
```

The data field `aList` is an instance of `ListInterface` and is a member of `SortedList`. The constructor and the `sortedAdd` method are implemented. The notation `aList.add` indicates an invocation to the insertion operation of the ADT list.

Programming Problem 1 at the end of this chapter asks you to complete this implementation. In doing so, you will realize that `locateIndex` needs `get` to access items in `aList`; that is, `ListInterface`'s implementation is hidden from `SortedList`. Notice also that a client of `SortedList` cannot access `aList` and has only the sorted list operations available.

9.5 Iterators

An iterator accesses a collection one item at a time

Earlier discussions about the Java Collections Framework introduced iterators. An iterator is an object that can access a collection of objects one object at a time. That is, an iterator traverses the collection of objects. Recall the JCF generic interface `java.util.ListIterator`:

```
public interface ListIterator<E> extends Iterator<E> {

  void add(E o);
  // Inserts the specified element into the list (optional
  // operation). The element is inserted immediately before
  // the next element that would be returned by next, if any,
  // and after the next element that would be returned by
  // previous, if any. (If the list contains no elements, the
  // new element becomes the sole element on the list.) The
  // new element is inserted before the implicit cursor: a
  // subsequent call to next would be unaffected, and a
  // subsequent call to previous would return the new element.

  boolean hasNext();
  // Returns true if this list iterator has more elements when
  // traversing the list in the forward direction.

  boolean hasPrevious();
  // Returns true if this list iterator has more elements when
  // traversing the list in the reverse direction.
```

```
    E next() throws NoSuchElementException;
    // Returns the next element in the list. Throws
    // NoSuchElementException if the iteration has no next
    // element.

    int nextIndex();
    // Returns the index of the element that would be returned
    // by a subsequent call to next. (Returns list size if the
    // list iterator is at the end of the list.)

    E previous()throws NoSuchElementException;
    // Returns the previous element in the list. This method may
    // be called repeatedly to iterate through the list
    // backwards, or intermixed with calls to next to go back
    // and forth. (Note that alternating calls to next and
    // previous will return the same element repeatedly.) Throws
    // NoSuchElementException if the iteration has no previous
    // element.

    int previousIndex()
    // Returns the index of the element that would be returned
    // by a subsequent call to previous. (Returns -1 if the list
    // iterator is at the beginning of the list.)

    void remove() throws UnsupportedOperationException,
                         IllegalStateException;
    // Removes from the list the last element that was
    // returned by next or previous (optional
    // operation).

    void set(E o) throws UnsupportedOperationException,
                         IllegalStateException;
    // Replaces the last element returned by next or
    // previous with the specified element (optional
    // operation).
} // end ListIterator
```

Some iterator methods may not be supported

Notice that many of the operations, such as *remove*, can throw the exception *UnsupportedOperationException*. The expectation is that operations will simply throw this exception if the operation is not available in the class that implements the interface.

An iterator has an implicit cursor that keeps track of where you are in the ADT. It is best to think of this cursor as being either before the first item in the list, between two items in the list, or after the last item in the list. The *hasPrevious* and the *previous* operations refer to the element before the cursor, whereas the *hasNext* and *next* operations refer to the element after the cursor. The *previous*

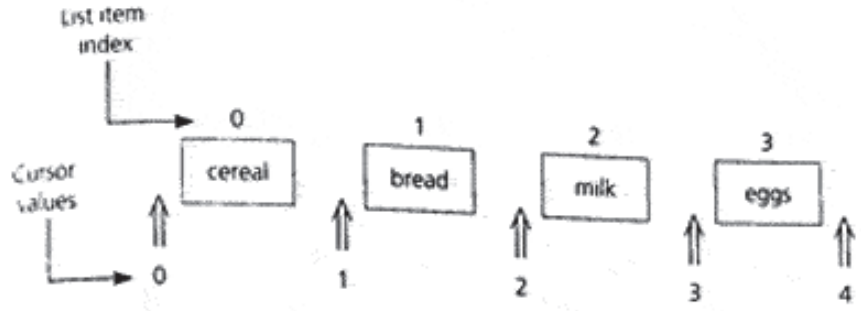


FIGURE 9-11
Relationship between iterator cursor and list elements

operation moves the cursor backward, whereas *next* moves it forward. Figure 9-11 shows the relationship between the iterator's cursor and the items in a *ListInterface* list. Note that the cursor value ranges from 0 to the size of the list.

So the implementation of an iterator must use some mechanism to keep track of the cursor. You also need to be able to identify the element that was extracted by the most recent call to *next* or *previous*. This is necessary for the implementation of the *remove* and *set* operations, which must operate on the last item returned by *previous* or *next*. And lastly, note that the operations *set* and *remove* may only be executed after a call to *previous* or *next*.

The implementation of an iterator can be approached in two ways. First, the iterator can be implemented using only the public methods available in the ADT. This leaves the iterator independent of the underlying implementation, but may not necessarily be the most efficient. So a second possibility is to implement the class as part of the ADT package, utilizing the underlying storage structure of the corresponding ADT to implement the iterator.

The following implementation is based upon the first approach, so it uses only the methods available in *ListInterface* to implement the *ListIterator* operations.

```
public class MyListIterator<T>
      implements java.util.ListIterator<T> {

  private ListInterface<T> list;
  private int cursor;          // location of the cursor in list
  private int lastItemIndex;   // index of last item returned
                               // by previous or next

  public MyListIterator(ListInterface<T> list) {
    this.list = list;
    cursor = 0;
    lastItemIndex = -1;
  } // end constructor
```

```
public void add(T item) {
  list.add(cursor+1, item);
  cursor++;
  lastItemIndex = -1;
} // end add

public boolean hasNext() {
  return (cursor < list.size());
} // end hasNext

public boolean hasPrevious() {
  return (cursor >= 0);
} // end hasPrevious

public T next() throws java.util.NoSuchElementException {
  try {
    T item = list.get(cursor + 1);
    lastItemIndex = cursor;
    cursor++;
    return item;
  } // end try
  catch (IndexOutOfBoundsException e){
    throw new java.util.NoSuchElementException();
  } // end catch
} // end next

public int nextIndex() {
  return cursor;
} // end nextIndex

public T previous()
       throws java.util.NoSuchElementException {
  try {
    T item = list.get(cursor);
    lastItemIndex = cursor;;
    cursor--;
    return item;
  } // end try
  catch(IndexOutOfBoundsException e) {
    throw new java.util.NoSuchElementException();
  } // catch
} // end previous

public int previousIndex() {
  return cursor - 1;
} // end previousIndex
```

```
public void remove() throws  UnsupportedOperationException,
                             IllegalStateException {
    See Programming Problem 10.
  throw new UnsupportedOperationException();
}  end remove

public void set(T item)
      throws UnsupportedOperationException {
    See Programming Problem 10.
  throw new UnsupportedOperationException();
}  end set

}  end MyListIterator
```

Note that if this iterator implementation is used with a `ListReferenceBased` list, this will be quite inefficient (Exercise 16 will ask you to explore this). As such, Programming Problem 11 asks you to create an iterator for a generic doubly linked list.

Summary

1. Classes can have ancestor and descendant relationships. A subclass inherits all members of its previously defined superclass, but can access only the public and protected members. Private members of a class are accessible only by its methods. Protected members can be accessed by methods of both the class and any subclasses, but not by clients of these classes.

2. With inheritance, the public and protected members of the superclass remain, respectively, public and protected members of the subclass. Such a subclass is type-compatible with its superclass. That is, you can use an instance of a subclass wherever you can use an instance of its superclass. This relationship between superclasses and subclasses is an *is-a* relationship.

3. A method in a subclass overrides a method in the superclass if they have the same parameter declarations. The Java annotation `@Override` provides a mechanism for a programmer to explicitly notify the compiler that a method from the superclass is being overridden. If the superclass has defined the method to be `final`, the subclass cannot override the method.

4. An abstract method in a class is a method that you can override in a subclass. When a method is abstract, you can either implement it or defer it to a further subclass.

5. A subclass inherits the interface of each method that is in its superclass. A subclass also inherits the implementation of each nonabstract method that is in its superclass.

6. An abstract class specifies only the essential members necessary for its subclasses and, therefore, can serve as the superclass for a family of classes. A class with at least one abstract method must also be declared abstract and is referred to as an abstract superclass.

7. Early, or static, binding describes a situation whereby a compiler can determine at compilation time the correct method to invoke. Late, or dynamic, binding describes a situation whereby the system makes this determination at execution time.
8. When you invoke a method that is not declared `final`—for example, `mySphere.displayStatistics()`—the type of object is the determining factor under late binding.
9. Generic classes enable you to parameterize the type of a class's data.
10. Iterators provide an alternative way to cycle through a collection of items.

Cautions

1. If a method is abstract, and an implementation is not provided by the subclass, the subclass must also be declared abstract.
2. If a class fails to implement all of the methods in an interface, it must be declared abstract.
3. You should use inheritance only if the relationship between two classes is *is-a*.

Self-Test Exercises

Self-Test Exercises 1, 2, and 3 consider the classes `Sphere` and `Ball`, which this chapter describes in the section "Inheritance Revisited."

1. Write Java statements for the following tasks:
   a. Declare an instance `mySphere` of `Sphere` with a radius 2.
   b. Declare an instance `myBall` of `Ball` whose radius is 6 and whose name is *Beach ball*.
   c. Display the diameters of `mySphere` and `myBall`.
2. Define a class `Planet` that inherits `Ball`, as defined in this chapter. Your new class should have private data fields that specify a planet's mimimum and maximum distances from the sun and public methods that access or alter these distances.
3. a. Can `resetBall`, which is a method of `Ball`, access `Sphere`'s data field `radius` directly, or must `resetBall` call `Sphere`'s `setRadius`? Explain.
   b. Repeat Part *a*, but assume that `radius` is a protected data field instead of a private data field of the class `Sphere`.
4. Consider the interface `SortedListInterface`, as described in this chapter.
   a. When can a reference be declared as type `SortedListInterface`?
   b. When must a class that implements `SortedListInterface` be used?
   c. Show a legal declaration that uses the `SortedListInterface` with one of the `SortedList` implementations.

5 a. What are the similarities between abstract classes and interfaces?

b. What are the differences between abstract classes and interfaces?

6 Why should a class's private methods never be abstract?

7 Given the generic class *NewClass* as described in the section "Generic Classes" write a statement that defines an instance *myClass* of *NewClass* for data declared as *Student*:

```
class Student {
  private String name;
  private double gpa;

  public Student(String name, double gpa) {
    this.name = name;
    this.gpa = gpa;
  } // end constructor
  ...
} // end Student
```

Exercises

1 Recall the classes *Sphere* and *Ball*, as described in this chapter in the section "Inheritance Revisited," and consider the following variation:

```
public class Sphere {
. . .
  public double area() { // surface area
    . . .
  } // end area

  public void displayStatistics() {
    . . .
  } // end displayStatistics
  . . .
} // end Sphere

class Ball extends Sphere {
  . . .
  public double area() { // cross-sectional area
  // Cross-sectional area is used to compute drag
  // on the ball
    . . .
  } // end area

  public void displayStatistics() {
    . . .
  } // end displayStatistics
  . . .
} // end Ball
```

Suppose that different implementations of *displayStatistics* appear in both *Sphere* and *Ball* and they each invoke the method *area*. Also assume that *b1* and *s1* are declared as follows:

```
Sphere s1 = new Sphere();
Ball b1 = new Ball();
```

For each of the scenarios shown next, indicate if the assignments are legal or illegal. In addition, if the assignments are legal, also explain which version of *area* each call to *displayStatistics* invokes. Be sure to explain your answer. Assume that each set of statements is independent and begins with the previous declarations.

a.
```
s1.displayStatistics();
b1.displayStatistics();
```

b.
```
s1.displayStatistics();
s1 = b1;
s1.displayStatistics();
b1.displayStatistics();
```

c.
```
b1.displayStatistics();
b1 = s1;
s1.displayStatistics();
b1.displayStatistics();
```

2. Define and implement a class *Pen* that has an instance of *Ball* as one of its data fields. Provide several members for the class *Pen*, such as the data field *color* and methods *isEmpty* and *write*.

3. Consider the following classes:

   Clock represents a device that keeps track of the time. Its public methods include *setTime* and *chime*.

   AlarmClock represents a clock that also has an alarm that can be set. Its public methods include *setSoundLevel* and *getAlarmTime*.

   a. Which of the methods mentioned above can the implementation of *setTime* invoke?

   b. Which of the methods mentioned above can the implementation of *getAlarmTime* invoke?

4. Assume the classes described in the previous question and consider a main method that contains the following statements:

```
Clock wallClock;
AlarmClock myAlarm;
```

   a. Which of these objects can correctly invoke the method *chime*?

   b. Which of these objects can correctly invoke the method *setSoundLevel*?

5. The `Node<T>` class used in the generic implementation of the class `ListReferencebased<T>` in this chapter assumed that it would be declared package-private; hence, the data fields were declared for package access only.

   a. Suppose that the data fields were declared private. Write accessor and mutator methods for both the `item` and `next` fields.

   b. Give at least three different examples of how the code in the `ListReferencebased<T>` implementation would have to be changed.

6. The section "Abstract Classes" gives a version of the abstract class `EquidistantShape` that contains a private data field `radius`.

   a. Define a class of circles as a subclass of `EquidistantShape`. Include additional methods for `diameter` and `circumference`.

   b. Revise the abstract class `EquidistantShape` so that `radius` is a protected data field instead of a private data field. Which methods are abstract and which methods must you implement?

   c. Repeat Part *a*, assuming the revision you made in Part *b*.

7. Consider the following classes:

   `Expression` represents algebraic expressions, including prefix, postfix, and infix expressions. Its public methods include `characterAt`. Its protected methods include `isOperator` and `isIdentifier`. It also has several private methods.

   `InfixExpression` is derived from `Expression` and represents infix expressions. Its public methods include `isLegal` and `evaluate`. It also has several protected and private methods.

   a. What methods can the implementation of `isIdentifier` invoke?

   b. What methods can the implementation of `isLegal` invoke?

8. Assume the classes described in the previous question and consider a main method that contains

```
Expression algExp;
InfixExpression infixExp;
```

   a. Which of these objects can correctly invoke the method `characterAt`?

   b. Which of these objects can correctly invoke the method `isOperator`?

   c. Which of these objects can correctly invoke the method `isLegal`?

9. Consider an ADT back list, which restricts insertions, deletions, and retrievals to the last item in the list.

   a. Define a generic interface `BackListInterface` that is a subclass of the interface `BasicListInterface`. Provide a reference-based implementation of the interface `BackListInterface` called `BackList`.

   b. Define and implement a class for the ADT stack that is a subclass of `BackList`.

10. Define an abstract class `Person` that describes a typical person. Include methods to retrieve the person's name, and to get or change his or her address. Next, define a subclass `Student` that describes a typical student. Include methods to retrieve his or her ID number, number of credits completed, and grade point average. Also include methods to get or change his or her campus address. Finally, derive from `Student` a class `UgradStudent` for a typical undergraduate student. Include methods for retrieving his or her degree and major.

11. Implement a generic version of `ListArrayBased` based on the generic `ListInterface` presented in this chapter. You will need to use the JFC `ArrayList` class instead of an array for the underlying implementation, since you cannot declare an array using a generic type in Java. Write a small test program as well to verify that your generic `ListArrayBased` class is working properly.

12. Complete the implementation of the class `ListReferenceBased` presented in the section "Generic Implementation of the Class List." In particular, write the implementations of the methods `get`, `add`, and `remove`.

13. Design and implement the following classes:

    a. An abstract class `Employee` that represents a generic employee. Include methods to retrieve information about an employee.

    b. A subclass of `Employee` called `HourlyEmployee` that describes an employee who gets paid by the hour. Include a public method called `pay()` that returns the pay of the employee for that week, and any other relevant methods to manage data fields such as hourly wage and number of hours worked.

    c. A subclass of `Employee` called `SalariedEmployee` that describes an employee who gets paid a fixed salary every month. Include a public method called `pay()` that returns the pay of the employee for that month, and any other relevant methods to manage data fields such as yearly salary and any unpaid leave taken that month.

14. Consider the following classes declared in the same package:

```
public abstract class Account {
  private String name;
  private String acctNum;
  protected double balance;
  String taxID;

  public Account(String name, String num) {
    this.name = name;
    acctNum = num;
  } // end constructor

} // end Account

public class SavingsAcct extends Account {
  double interestRate;

  public SavingsAcct(String name, String num) {
    // See part a)
    // and constructor
  } // end SavingsAcct
```

```
public class CheckingAcct extends Account {
  double checkFee;
  private double overdraftLimit;

  public CheckingAcct(String name, String num, double fee) {
    // See part b)
  } // end constructor
} // end CheckingAcct

public class Stock {
  private String stockCode;
  int numShares;
} // end Stock
```

a. Write code for the constructor that could be used to initialize a `SavingsAcct` object.

b. Write code for the constructor that could be used to initialize a `CheckingAcct` object.

For the following code snippets, which are contained in the same package as the classes just given, identify whether the code

[A] fails to compile,

[B] compiles with a warning,

[C] generates an error at runtime, or

[D] compiles and runs without problem.

c.
```
CheckingAcct c = new CheckingAcct("Janet", "JP4561", 0.5);
c.taxID = "987-65-4320";
c.acctNum = "123456";
```

d.
```
Account a = new SavingsAcct("Diane", "DG0621");
a.balance = 10.00;
```

e.
```
Account s = new Stock();
s.numShares = 2500;
```

f.
```
Account a = new Account("Marlene", "MP0108");
a.balance = 2500;
```

g.
```
CheckingAcct c = new CheckingAcct("Karen", "KM0312", 0.5);
Account a = c;
a.checkFee = 0.75;
```

h.
```
SavingsAcct s = new Account("Mike", "MH0621");
s.interestRate = 1.5;
```

15. Assume the following classes added to the same package as the classes presented in Exercise 14.

```
import java.util.*;
Public class Portfolio<T> {
  Private ArrayList<T> accounts = new ArrayList<T>();
  Private int numAccts = 0;
```

```
    public void addAcct(T a) {
      accounts.add(numAccts, a);
      numAccts++;
    } // end addAcct
  } // end Portfolio
```

For the following code snippets, which are contained in the same package as the classes just given, identify whether the code

[A] fails to compile,

[B] compiles with a warning,

[C] generates an error at runtime,or

[D] compiles and runs without problem.

a.
```
Portfolio<Account> myAccts = new Portfolio<Account>();
myAccts.addAcct(new CheckingAcct("Kate", "KR1225", 0.5));
```

b.
```
Portfolio<Account> myAccts = new Portfolio<SavingsAcct>();
myAccts.addAcct(new SavingsAcct("Andrew", "AH0527"));
```

c.
```
Portfolio<Account> myAccts = new Portfolio<Account>();
myAccts.addAcct(new Stock());
```

d.
```
Portfolio<? extends Account> acc1;
Portfolio<SavingsAcct> acc2 = new Portfolio<SavingsAcct>();
acc1 = acc2;
```

e.
```
Portfolio myAccts = new Portfolio();
myAccts.addAcct(new CheckingAcct("Sarah", "SH0604",0.5));
```

16. It was noted that the *MyListIterator* implementation presented in section 9.5 would be inefficient if it were used with a *ListReferenceBased* list. Explain why this is the case and give a specific example demonstrating this inefficiency.

Programming Problems

1. Complete the implementation of *SortedList* that has a reference of type *ListInterface* as a data field, as described in the section "Implementations of the ADT Sorted List that Use the ADT List."

2. The interface *ListInterface*, as described in this chapter, does not contain a method *position* that returns the number of a particular item, given the item's value. Such a method enables you to pass the node's number to *remove*, for example, to delete the item.

   Define a subinterface of *ListInterface* that has *position* as a method as well as methods that insert, delete, and retrieve items by their values instead of their positions. Write a class that implements this interface. Always make insertions at the beginning of the list. Although the items in this list are not sorted, the new ADT is analogous to *SortedList*, which contains the method *locatePosition*.

3. Consider an ADT circular list, which is like the ADT list but treats its first item as being immediately after its last item. For example, if a circular list contains six items, retrieval or deletion of the eighth item actually involves the list's second item. Let insertion into a circular list, however, behave exactly like insertion into a list. Define and implement the ADT circular list as a subclass of `ListReferenceBased`, as described in Chapter 5.

4. Programming Problem 12 in Chapter 7 describes the ADT traversable stack. In addition to the standard stack operations—`isEmpty`, `push`, `pop`, and `peek`—a traversable stack includes the operation `traverse`. The `traverse` operation is an iterator that begins at the bottom of the stack and displays each item in the stack until it reaches the top of the stack.

   Alternatively, you could create a `StackIterator` class based on the `StackInterface`, as given in Chapter 7. However, the iterator would then only have access to the methods contained in the interface, such as `push`, `pop`, and `peek`. Since the iterator should not change the contents of the stack, the implementation would be quite inefficient.

   The `StackIterator` class would be more efficient if it had access to the underlying stack implementation (either `StackArrayBased` or `StackReferenceBased`). Implement a `Stack` class within a package that provides package access to the stack's underlying data structure. Then, in the same package, create a `StackIterator` class that uses this new `Stack` class.

5. As discussed in Chapter 8, the JCF provides an interface for a generic double-ended queue which supports insertion and deletion of items from both the front and back of the data structure. Here is a simplified version of a generic deque interface:

```
public interface Deque<E> {

  boolean isEmpty();
  // Return true if the deque is empty, false otherwise.

  boolean addFirst(E item);
  // Insert the item at the front of the deque. Returns false
  // if the item cannot be added to the deque, true otherwise.

  boolean addLast(E item);
  // Insert the item at the back of the deque. Returns false if
  // the item cannot be added to the deque, true otherwise.

  boolean contains(Object o);
  // Returns true if this deque contains the specified el

  E removeFirst();
  // Delete and return the fir
  // is not empty, otherwise

  E removeLast();
  // Delete and
  // is not emp
```

```
    E peekFirst();
    // Return the first item in the deque if the deque is
    // not empty, leaving the deque unchanged. Otherwise return
    // null (the deque was empty).

    E peekLast();
    // Return the last item in the deque if the deque is
    // not empty, leaving the deque unchanged. Otherwise return
    // null (the deque was empty).
} // end Deque
```

a. Create a reference-based implementation of this generic `Deque` interface.

b. Create an array-based implementation of this generic `Deque` interface using `ArrayList`.

6. Write a generic version of the class `Stack` presented in Chapter 7.

   a. Use the reference-based version as the basis of your implementation.

   b. Use the array-based version as the basis of your implementation, using an `ArrayList` for the array.

7. Write a generic version of the class `Queue` presented in Chapter 8.

   a. Use the reference-based version as the basis of your implementation.

   b. Use the array-based version as the basis of your implementation, using an `ArrayList` for the array.

8. Algebraic expressions are character strings, but since `String` is a `final` class, you cannot derive a class of expressions from `String`. Instead, define an abstract class `Expression` that can be the basis of other classes of algebraic expressions. Provide methods such as `characterAt`, `isOperator`, and `isIdentifier`. Next design and implement a class `InfixExpression` derived from the class `Expression`.

   Programming Problem 9 of Chapter 6 describes a grammar and a recognition algorithm for infix algebraic expressions. That grammar makes left-to-right association illegal when consecutive operators have the same precedence. Thus, *a/b*c* is illegal, but both *a/(b*c)* and *(a/b)*c* are legal.

   Programming Problem 8 of Chapter 7 describes an algorithm to evaluate an infix expression that is syntactically correct by using two stacks.

   Include in the class `InfixExpression` an `isLegal` method—based on the recognition algorithm given in Chapter 6—and an `evaluate` method—based on the evaluation algorithm given in Chapter 7. Use one of the ADT stack implementations presented in Chapter 7.

9. Chapter 6 described the class `Queens` that was used in a solution to the Eight Queens problem. A two-dimensional array represented the chessboard and was a data field of the class. Programming Problem 1 of Chapter 6 asked you to write a program to solve the Eight Queens problem based on these ideas. Revise that program by replacing the two-dimensional array with a class that represents the chessboard.

10 Implement the methods `remove` and `set` for the class `MyListIterator`. These methods should behave as follows:

`public void remove()`

Removes from the list the last element that was returned by `next` or `previous`. This call can be made only once per call to `next` or `previous`. It can be made only if the `add` method in `MyListIterator` has not been called after the last call to `next` or `previous`.

Throws `IllegalStateException` if neither `next` nor `previous` has been called, or if `remove` or `add` has been called after the last call to `next` or `previous`.

`public void set(T item)`

Replaces the last element returned by `next` or `previous` with the specified element. This call can be made only if neither `remove` nor `add` have been called after the last call to `next` or `previous`.

Throws `IllegalStateException` if neither `next` nor `previous` has been called, or if `remove` or `add` has been called after the last call to `next` or `previous`.

Note that `remove` and `set` require you to keep track of the state of the iterator; in other words, you must know whether or not `next` and `previous` are called immediately before the use of `remove` or `set`. You may also need to keep track of whether `previous` or `next` was called last to make sure that the correct element is deleted from or replaced in the list when `remove` and `set` are called. You may find it useful to use the variable `lastItemIndex` to determine if the last call made was to `next` or `previous`.

11 Implement a generic doubly linked list implementation called `DoubleRefBasedList` that implements the generic `ListInterface` presented in this chapter. In addition, create an iterator called `DListIterator` for the `DoubleRefBasedList` class. To make the implementation of `DListIterator` more efficient, you need to have access to the underlying doubly linked list in `DoubleRefBasedList`. The easiest way to accomplish this is to move the iterator class inside the list class. This inner class is then a member of the `DoubleRefBasedList` class and will have access to all of the members of that class. So the `DoubleRefBasedList` class with an inner `DListIterator` class will be structured as follows:

```
public class DoubleRefBasedList<T> implements ListInterface<T> {

  private DNode<T> head;
  // Assume Dnode has been defined as a node class with both
  // a next and previous reference.  It is used to implement
  // the doubly linked list.

  // Class members for ListInterface implementation appear here.

  // Inner class DListIterator
  private class DListIterator
          implements java.util.ListIterator<T> {

    private DNode<T> cursor = head;
```

```
    // Class members for ListIterator implementation appear here.
    // This inner class has access to members of the outer class.

  } // end DListIterator

  public java.util.ListIterator<T> listIterator() {
    return new DListIterator();
  } // end listIterator
} // end DoubleRefBasedList
```

The cursor declaration is shown to demonstrate how members of the outer class can be accessed from the inner class. When a call is made to the *DoubleRefBasedList* method *listIterator*, the cursor will be initialized to reference the first node in the list.

Also write a test class that tests that the operations for both the *DoubleRefBasedList* class and the *DListIterator* class are working properly.

12. Create a generic class for a circular doubly linked list as shown in Figure 5-29 of Chapter 5. Also define and implement a bidirectional iterator for this class.

CHAPTER 10

Algorithm Efficiency and Sorting

This chapter will show you how to analyze the efficiency of algorithms. The basic mathematical techniques for analyzing algorithms are central to more advanced topics in computer science and give you a way to formalize the notion that one algorithm is significantly more efficient than another. As examples, you will see analyses of some algorithms that you have studied before, including those that search data. In addition, this chapter examines the important topic of sorting data. You will study some simple algorithms, which you may have seen before, and some more-sophisticated recursive algorithms. Sorting algorithms provide varied and relatively easy examples of the analysis of efficiency.

10.1 Measuring the Efficiency of Algorithms

The comparison of algorithms is a topic that is central to computer science. Measuring an algorithm's efficiency is quite important because your choice of algorithm for a given application often has a great impact. Responsive word processors, grocery checkout systems, automatic teller machines, video games, and life support systems all depend on efficient algorithms.

Suppose two algorithms perform the same task, such as searching. What does it mean to compare the algorithms and conclude that one is better? Chapter 2 discussed the several components that contribute to the cost of a computer program. Some of these components involve the cost of human time—the time of the people who develop, maintain, and use the program. The other components involve the cost of program execution—that is, the program's efficiency—measured by the amount of computer time and space that the program requires to execute.

We have, up to this point, emphasized the human cost of a computer program. The early chapters of this book stressed style and readability. They pointed out that well-designed algorithms reduce the human costs of implementing the algorithm with a program, of maintaining the program, and of modifying the program. The primary concern has been to develop good problem-solving skills and programming style. Although we will continue to concentrate our efforts in that direction, the efficiency of algorithms is also important. Efficiency is one criterion that you should use when selecting an algorithm and its implementation. The solutions in this book, in addition to illustrating good programming style, are frequently based on relatively efficient algorithms.

Consider efficiency when selecting an algorithm

The **analysis of algorithms** is the area of computer science that provides tools for contrasting the efficiency of different methods of solution. Notice the use of the term *methods of solution* rather than *programs*; it is important to emphasize that the analysis concerns itself primarily with *significant* differences in efficiency—differences that you can usually obtain only through superior methods of solution and rarely through clever tricks in coding. Reductions in computing costs due to clever coding tricks are often more than offset by reduced program readability, which increases human costs. An analysis should focus on gross differences in the efficiency of algorithms that are likely to dominate the overall cost of a solution. To do otherwise could lead you to select an algorithm that runs a small fraction of a second faster than another algorithm yet requires many more hours of your time to implement and maintain.

A comparison of algorithms should focus on significant differences in efficiency

The efficient use of both time and memory is important. Computer scientists use similar techniques to analyze an algorithm's time and space efficiency. Since none of the algorithms covered in this text has significant space requirements, our focus will be primarily on time efficiency.

How do you compare the time efficiency of two algorithms that solve the same problem? One possible approach is to implement the two algorithms in Java and run the programs. This approach has at least three fundamental difficulties:

Three difficulties with comparing programs instead of algorithms

1. **How are the algorithms coded?** If algorithm A_1 runs faster than algorithm A_2, it could be the result of better programming. Thus, if you compare the running times of the programs, you are really comparing implementations of the algorithms rather than the algorithms themselves. You should not compare implementations, because they are sensitive to factors such as programming style that tend to cloud the issue of which algorithm is inherently more efficient.

2. **What computer should you use?** The particular computer on which the programs are run also obscures the issue of which algorithm is inherently more efficient. One computer may simply be much faster than the other, so clearly you should use the same computer for both programs. Which computer should you choose? The particular operations that the algorithms require can cause A_1 to run faster than A_2 on one computer, while the opposite is true on another computer. You should compare the efficiency of the algorithms independently of a particular computer.

3. **What data should the programs use?** Perhaps the most important difficulty on this list is the selection of the data for the programs to use. There is always the danger that you will select instances of the problem for which one of the algorithms runs uncharacteristically fast. For example, when comparing a sequential search and a binary search of a sorted array, you might search for an item that happens to be the smallest item in the array. In such a case, the sequential search will find the item more quickly than the binary search because the item is first in the array and so is the first item that the sequential search will examine. Any analysis of efficiency must be independent of specific data.

Algorithm analysis should be independent of specific implementations, computers, and data

To overcome these difficulties, computer scientists employ mathematical techniques that analyze algorithms independently of specific implementations, computers, or data. You begin this analysis by counting the number of significant operations in a particular solution, as the next section describes.

The Execution Time of Algorithms

Counting an algorithm's operations is a way to assess its efficiency

Previous chapters have informally compared different solutions to a given problem by looking at the number of operations that each solution required. For example, Chapter 5 compared array-based and reference-based implementations of the ADT list and found that an array-based `list.get(n)` could access the n^{th} item in a list directly in one step, because the item is stored in `items[n-1]`. A reference-based `list.get(n)`, however, must traverse the list from its beginning until the n^{th} node is reached, and so would require n steps.

An algorithm's execution time is related to the number of operations it requires. Counting an algorithm's operations—if possible—is a way to assess its efficiency. Consider a few other examples.

Traversal of a linked list. Recall from Chapter 5 that you can display the contents of a linked list that *head* references by using the following statements:[1]

```
Node curr = head;                          ← 1 assignment
while (curr != null) {                     ← n+1 comparisons
  System.out.println(curr.item);           ← n writes
  curr = curr.next;                        ← n assignments
}  // end while
```

Displaying the data in a linked list of n nodes requires time proportional to n

Assuming a linked list of n nodes, these statements require $n + 1$ assignments, $n + 1$ comparisons, and n write operations. If each assignment, comparison, and write operation requires, respectively, a, c, and w time units, the statements require $(n + 1) * (a + c) + n * w$ time units.[2] Thus, the time required to write n nodes is proportional to n. This conclusion makes sense intuitively: It takes longer to display, or traverse, a linked list of 100 items than it does a linked list of 10 items.

The Towers of Hanoi. Chapter 6 proved recursively that the solution to the Towers of Hanoi problem with n disks requires $2^n - 1$ moves. If each move requires the same time m, the solution requires $(2^n - 1) * m$ time units. As you will soon see, this time requirement increases rapidly as the number of disks increases.

Nested loops. Consider an algorithm that contains nested loops of the following form:

```
for (i = 1 through n) {
  for (j = 1 through i) {
    for (k = 1 through 5) {
      Task T
    }  // end for
  }  // end for
}  // end for
```

If task T requires t time units, the innermost loop on k requires $5 * t$ time units. The loop on j requires $5 * t * i$ time units, and the outermost loop on i requires

$$\sum_{i=1}^{n} (5 * t * i) = 5 * t * (1 + 2 + \cdots + n) = 5 * t * n * (n + 1)/2$$

time units.

---

1. Chapter 5 actually used a `for` statement. We use an equivalent `while` statement here to clarify the analysis.
2. Although omitting multiplication operators is common in algebra, we indicate them explicitly here to facilitate counting them.

Algorithm Growth Rates

Measures an algorithm's time requirement as a function of the problem size

As you can see, the previous examples derive an algorithm's time requirement as a function of the problem size. The way to measure a problem's size depends on the application—typical examples are the number of nodes in a linked list, the number of disks in the Tower of Hanoi problem, the size of an array, or the number of items in a stack. Thus, we reached conclusions such as

Algorithm A requires $n^2/5$ time units to solve a problem of size n

*Algorithm B requires 5 * n time units to solve a problem of size n*

The time units in these two statements must be the same before you can compare the efficiency of the two algorithms. Perhaps we should have written

Algorithm A requires $n^2/5$ seconds to solve a problem of size n

Our earlier discussion indicates the difficulties with such a statement: On what computer does the algorithm require $n^2/5$ seconds? What implementation of the algorithm requires $n^2/5$ seconds? What data caused the algorithm to require $n^2/5$ seconds?

What specifically do you want to know about the time requirement of an algorithm? The most important thing to learn is how quickly the algorithm's time requirement grows as a function of the problem size. Statements such as

Algorithm A requires time proportional to n^2

Algorithm B requires time proportional to n

each express an algorithm's proportional time requirement, or **growth rate**, and enable you to compare algorithm *A* with another algorithm *B*. Although you cannot determine the exact time requirement for either algorithm *A* or algorithm *B* from these statements, you can determine that for large problems, *B* will require significantly less time than *A*. That is, *B*'s time requirement—as a function of the problem size *n*—increases at a slower rate than *A*'s time requirement, because *n* increases at a slower rate than n^2. Even if *B* actually requires 5 * *n* seconds and *A* actually requires $n^2/5$ seconds, *B* eventually will require significantly less time than *A*, as *n* increases. Figure 10-1 illustrates this fact. Thus, an assertion like "*A* requires time proportional to n^2" is exactly the kind of statement that characterizes the inherent efficiency of an algorithm independently of such factors as particular computers and implementations.

Compare algorithm efficiencies for large problems

Figure 10-1 also shows that *A*'s time requirement does not exceed *B*'s until *n* exceeds 25. Algorithm efficiency is typically a concern for large problems only. The time requirements for small problems are generally not large enough to matter. Thus, our analyses assume large values of *n*.

Order-of-Magnitude Analysis and Big O Notation

If

Algorithm A requires time proportional to f(n)

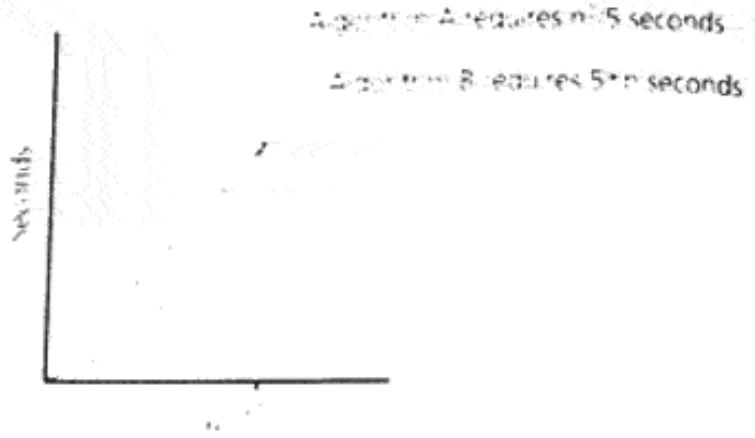


FIGURE 10-1

Time requirements as a function of the problem size

Algorithm A is said to be **order** $f(n)$, which is denoted as $O(f(n))$. The function $f(n)$ is called the algorithm's **growth rate function.** Because the notation uses the capital letter O to denote *order*, it is called the **Big O notation.** If a problem of size n requires time that is directly proportional to n, the problem is $O(n)$—that is, order n. If the time requirement is directly proportional to n^2, the problem is $O(n^2)$, and so on.

The following definition formalizes these ideas

KEY CONCEPTS

Definition of the Order of an Algorithm

Algorithm A is order $f(n)$—denoted $O(f(n))$—if constants k and n_0 exist such that A requires no more than $k * f(n)$ time units to solve a problem of size $n \geq n_0$.

The requirement $n \geq n_0$ in the definition of $O(f(n))$ formalizes the notion of sufficiently large problems. In general, many values of k and n can satisfy the definition.

The following examples illustrate the definition:

- Suppose that an algorithm requires $n^2 - 3 * n + 10$ seconds to solve a problem of size n. If constants k and n_0 exist such that

  $k * n^2 > n^2 - 3 * n + 10$ for all $n \geq n_0$

  the algorithm is order n^2. In fact, if k is 3 and n_0 is 2,

  $3 * n^2 > n^2 - 3 * n + 10$ for all $n \geq 2$

as Figure 10-2 illustrates. Thus, the algorithm requires no more than $k * n^2$ time units for $n \geq n_0$, and so is $O(n^2)$.

- Previously, we found that displaying a linked list's first n items requires $(n + 1) * (a + c) + n * w$ time units. Since $2 * n \geq n + 1$ for $n \geq 1$,

  $$(2 * a + 2 * c + w) * n \geq (n + 1) * (a + c) + n * w \text{ for } n \geq 1$$

  Thus, this task is $O(n)$. Here, k is $2 * a + 2 * c + w$, and n_0 is 1.

- Similarly, the solution to the Towers of Hanoi problem requires $(2^n - 1) * m$ time units. Since

  $$m * 2^n > (2^n - 1) * m \text{ for } n \geq 1$$

  the solution is $O(2^n)$.

The requirement $n \geq n_0$ in the definition of $O(f(n))$ means that the time estimate is correct for sufficiently large problems. In other words, the time estimate is too small for at most a finite number of problem sizes. For example, the function $\log n$ takes on the value 0 when n is 1. Thus, the fact that $k * \log 1$ is 0 for all constants k implies an unrealistic time requirement; presumably, all algorithms require more than 0 time units even to solve a problem of size 1. Thus, you can discount problems of size $n = 1$ if $f(n)$ is $\log n$.

To dramatize further the significance of an algorithm's proportional growth rate, consider the table and graph in Figure 10-3. The table (Figure 10-3a) gives, for various values of n, the approximate values of some common growth-rate functions, which are listed in order of growth:

$$O(1) < O(\log_2 n) < O(n) < O(n * \log_2 n) < O(n^2) < O(n^3) < O(2^n)$$

Order of growth of some common functions

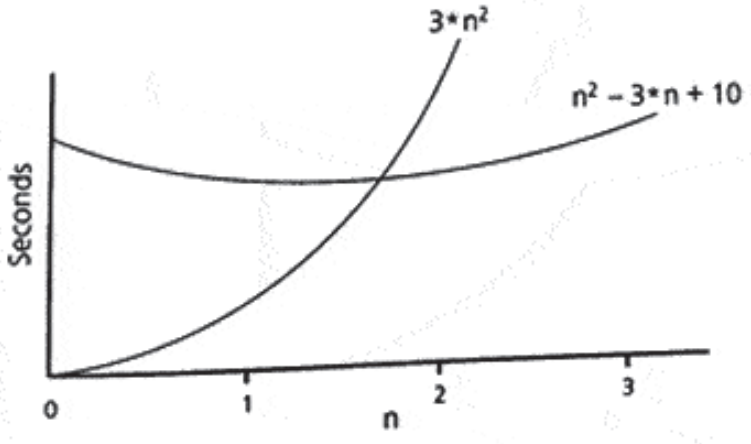


FIGURE 10-2

When $n \geq 2$, $3 * n^2$ exceeds $n^2 - 3 * n + 10$

The table demonstrates the relative speed at which the values of the functions grow. (Figure 10-3b represents the growth-rate functions graphically.[3])

(a)

	n					
Function	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

(b)

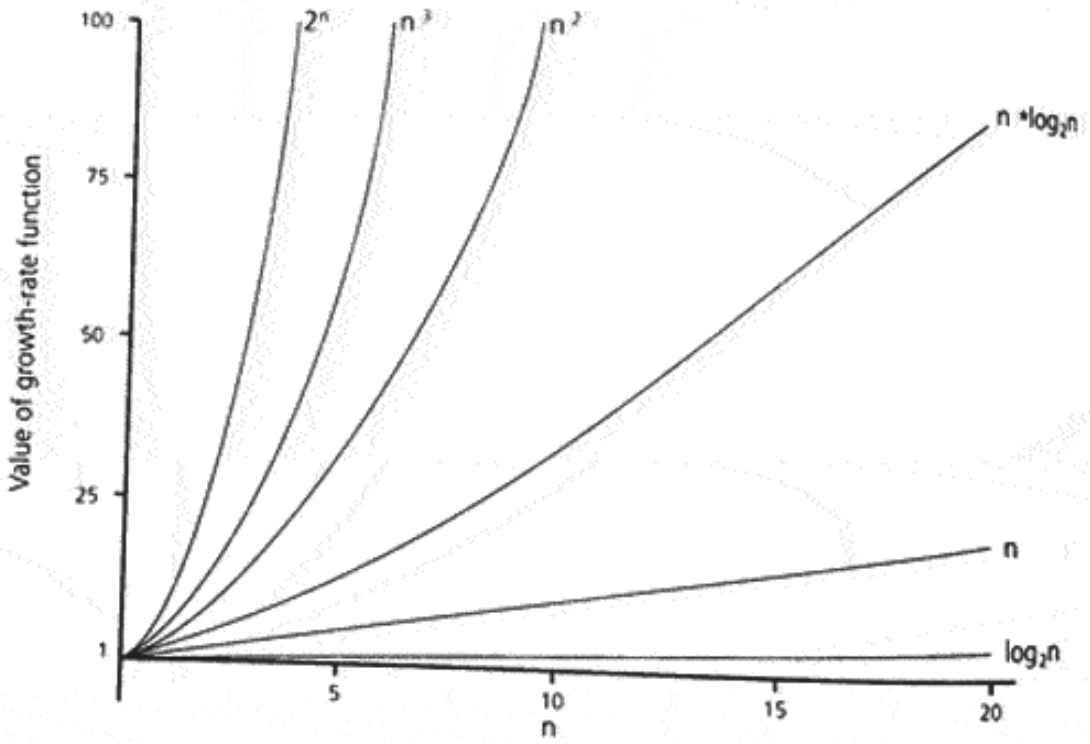


FIGURE 10-3

A comparison of growth-rate functions: (a) in tabular form; (b) in graphical form

3. The graph of $f(n) = 1$ is omitted because the scale of the figure makes it difficult to draw. It would, however, be a straight line parallel to the x axis through $y = 1$.

These growth-rate functions have the following intuitive interpretations:

Intuitive interpretations of growth-rate functions

1	A growth-rate function of 1 implies a problem whose time requirement is constant and, therefore, independent of the problem's size n.
$\log_2 n$	The time requirement for a **logarithmic** algorithm increases slowly as the problem size increases. If you square the problem size, you only double its time requirement. Later you will see that the recursive binary search algorithm that you studied in Chapter 3 has this behavior. Recall that a binary search halves an array and then searches one of the halves. Typical logarithmic algorithms solve a problem by solving a smaller constant fraction of the problem. The base of the log does not affect a logarithmic growth rate, so you can omit it in a growth-rate function. Exercise 6 at the end of this chapter asks you to show why this is true.
n	The time requirement for a **linear** algorithm increases directly with the size of the problem. If you square the problem size, you also square its time requirement.
$n * \log_2 n$	The time requirement for an $n * \log_2 n$ algorithm increases more rapidly than a linear algorithm. Such algorithms usually divide a problem into smaller problems that are each solved separately. You will see an example of such an algorithm—the mergesort—later in this chapter.
n^2	The time requirement for a **quadratic** algorithm increases rapidly with the size of the problem. Algorithms that use two nested loops are often quadratic. Such algorithms are practical only for small problems. Later in this chapter, you will study several quadratic sorting algorithms.
n^3	The time requirement for a **cubic** algorithm increases more rapidly with the size of the problem than the time requirement for a quadratic algorithm. Algorithms that use three nested loops are often cubic, and are practical only for small problems.
2^n	As the size of a problem increases, the time requirement for an exponential algorithm usually increases too rapidly to be practical.

If algorithm A requires time that is proportional to function f and algorithm B requires time that is proportional to a slower growing function g, it is apparent that B will always be significantly more efficient than A for large enough problems. For large problems, the proportional growth rate dominates all other factors in determining an algorithm's efficiency.

Several mathematical properties of Big O notation help to simplify the analysis of an algorithm. As we discuss these properties, you should keep in mind that $O(f(n))$ means "is of order $f(n)$" or "has order $f(n)$." O is not a function.

Some properties of growth-rate functions

1. **You can ignore low-order terms in an algorithm's growth-rate function.** For example, if an algorithm is $O(n^3 + 4 * n^2 + 3 * n)$, it is also $O(n^3)$. By examining the table in Figure 10-3a, you can see that the n^3 term is significantly larger than either $4 * n^2$ or $3 * n$, particularly for large values of n. For large n, the growth rate of $n^3 + 4 * n^2 + 3 * n$ is the same as the growth rate of n^3. It is the growth rate of $f(n)$, not the value of $f(n)$, that is important here. Thus, even if an algorithm is $O(n^3 + 4 * n^2 + 3 * n)$, we say that it is simply $O(n^3)$. In general, you can usually conclude that an algorithm is $O(f(n))$, where f is a function similar to the ones listed in Figure 10-3.

2. **You can ignore a multiplicative constant in the high-order term of an algorithm's growth-rate function.** For example, if an algorithm is $O(5 * n^3)$, it is also $O(n^3)$. This observation follows from the definition of $O(f(n))$, if you let $k = 5$.

3. $O(f(n)) + O(g(n)) = O(f(n) + g(n))$. You can combine growth-rate functions. For example, if an algorithm is $O(n^2) + O(n)$, it is also $O(n^2 + n)$, which you write simply as $O(n^2)$ by applying property 1. Analogous rules hold for multiplication.

These properties imply that you need only an estimate of the time requirement to obtain an algorithm's growth rate; you do not need an exact statement of an algorithm's time requirement, which is fortunate because deriving the exact time requirement is often difficult and sometimes impossible.

An algorithm can require different times to solve different problems of the same size

Worst-case and average-case analyses. A particular algorithm might require different times to solve different problems of the same size. For example, the time that an algorithm requires to search n items might depend on the nature of the items. Usually you consider the maximum amount of time that an algorithm can require to solve a problem of size n—that is, the worst case. **Worst-case analysis** concludes that algorithm A is $O(f(n))$ if, in the worst case, A requires no more than $k * f(n)$ time units to solve a problem of size n for all but a finite number of values of n. Although a worst-case analysis can produce a pessimistic time estimate, such an estimate does not mean that your algorithm will always be slow. Instead, you have shown that the algorithm will never be slower than your estimate. Realize, however, that an algorithm's worst case might happen rarely, if at all, in practice.

An **average-case analysis** attempts to determine the average amount of time that an algorithm requires to solve problems of size n. In an average-case analysis, A is $O(f(n))$ if the average amount of time that A requires to solve a problem of size n is no more than $k * f(n)$ time units for all but a finite number of values of n. Average-case analysis is, in general, far more difficult to perform than worst-case analysis. One difficulty is determining the relative probabilities of encountering various problems of a given size; another is determining the distributions of various data values. Worst-case analysis is easier to calculate and is thus more common.

Keeping Your Perspective

Before continuing with additional order-of-magnitude analyses of specific algorithms, a few words about perspective are appropriate. For example, consider an ADT list of n items. You saw earlier that an array-based `list.get(n)` operation can access the n^{th} item in the list directly. This access is independent of n. `get` takes the same time to access the hundredth item as it does to access the first item in the list. Thus, the array-based implementation of the retrieval operation is $O(1)$. However, the reference-based implementation of `get` in the class `ListReferenceBased` requires n steps to traverse the list until it reaches the n^{th} item, and so is $O(n)$.

An array-based **get** is O(1)

A reference-based **get** is O(n)

Throughout the course of an analysis, you should always keep in mind that you are interested only in *significant* differences in efficiency. Is the difference in efficiency for the two implementations of `get` significant? As the size of the list grows, the reference-based implementation might require more time to retrieve the desired node, because the node can be farther away from the beginning of the list. In contrast, regardless of how large the list is, the array-based implementation always requires the same constant amount of time to retrieve any particular item. Thus, no matter what your notion of a significant difference in time is, you will reach this time difference if the list is large enough. In this example, observe that the difference in efficiency for the two implementations is worth considering only when the problem is large enough. If the list never has more than 25 items, for example, the difference in the implementations is not significant at all.

When choosing an ADT's implementation, consider how frequently particular ADT operations occur in a given application

Now consider an application—such as a word processor's spelling checker—that frequently retrieves items from a list but rarely inserts or deletes an item. Since an array-based `get` is faster than a reference-based `get`, you should choose an array-based implementation of the list for the application. On the other hand, if an application requires frequent insertions and deletions but rarely retrieves an item, you should choose a reference-based implementation of the list. The most appropriate implementation of an ADT for a given application strongly depends on how frequently the application will perform the operations. You will see more examples of this point in the next chapter.

Some seldom-used but critical operations must be efficient

The response time of some ADT operations, however, can be crucial, even if you seldom use them. For example, an air traffic control system could include an emergency operation to resolve the imminent collision of two airplanes. Clearly,

this operation must occur quickly, even if it is rarely used. Thus, before you choose an implementation for an ADT, you should know what operations a particular application requires, approximately how often the application will perform each operation, and the response times that the application requires of each operation.

Soon we will compare a searching algorithm that is $O(n)$ with one that is $O(\log_2 n)$. While it is true that an $O(\log_2 n)$ searching algorithm requires significantly less time on large arrays than an $O(n)$ algorithm requires, on small arrays—say, $n < 25$—the time requirements might not be significantly different at all. In fact, it is entirely possible that, because of factors such as the size of the constant k in the definition of Big O, the $O(n)$ algorithm will run faster on small problems. It is only on large problems that the slower growth rate of an algorithm necessarily gives it a significant advantage. Figure 10-1 illustrated this phenomenon.

If the problem size is always small, you can probably ignore an algorithm's efficiency

Thus, in general, if the maximum size of a given problem is small, the time requirements of any two solutions for that problem likely will not differ significantly. If you know that your problem size will always be small, do not overanalyze; simply choose the algorithm that is easiest to understand, verify, and code.

Weigh the trade-offs between an algorithm's time requirements and its memory requirements

Frequently, when evaluating an algorithm's efficiency, you have to weigh carefully the trade-offs between a solution's execution time requirements and its memory requirements. You are rarely able to make a statement as strong as "This method is the best one for performing the task." A solution that requires a relatively small amount of computer time often also requires a relatively large amount of memory. It may not even be possible to say that one solution requires less time than another. Solution *A* may perform some components of the task faster than solution *B*, while solution *B* performs other components of the task faster than solution *A*. Often you must analyze the solutions in light of a particular application.

Compare algorithms for both style and efficiency

In summary, it is important to examine an algorithm for both style and efficiency. The analysis should focus only on gross differences in efficiency and not reward coding tricks that save milliseconds. Any finer differences in efficiency are likely to interact with coding issues, which you should not allow to interfere with the development of your programming style. If you find a method of solution that is significantly more efficient than others, you should select it, unless you know that the maximum problem size is quite small. If you will be solving only small problems, it is possible that a less efficient algorithm would be more appropriate. That is, other factors, such as the simplicity of the algorithm, could become more significant than minor differences in efficiency.

Order-of-magnitude analysis focuses on large problems

In fact, performing an order-of-magnitude analysis implicitly assumes that an algorithm will be used to solve large problems. This assumption allows you to focus on growth rates because, regardless of other factors, an algorithm with a slow growth rate will require less time than an algorithm with a fast growth rate, provided that the problems to be solved are sufficiently large.

The Efficiency of Searching Algorithms

As another example of order-of-magnitude analysis, consider the efficiency of two search algorithms: the sequential search and the binary search of an array.

Sequential search. In a sequential search of an array of n items, you look at each item in turn, beginning with the first one, until either you find the desired item or you reach the end of the data collection. In the best case, the desired item is the first one that you examine, so only one comparison is necessary. Thus, in the best case, a sequential search is O(1). In the worst case, the desired item is the last one you examine, so n comparisons are necessary. Thus, in the worst case, the algorithm is O(n). In the average case, you would find the desired item in the middle of the collection, making $n/2$ comparisons. Thus, the algorithm is O(n) in the average case.

Sequential search
Worst case O(n).
average case O(n).
best case O(1)

What is the algorithm's order when you do not find the desired item? Does the algorithm's order depend on whether or not the initial data is sorted? These questions are left for you in Self-Test Exercise 4 at the end of this chapter.

Binary search. Is a binary search of an array more efficient than a sequential search? The binary search algorithm, which Chapter 3 presents, searches a sorted array for a particular item by repeatedly dividing the array in half. The algorithm determines which half the item must be in—if it is indeed present—and discards the other half. Thus, the binary search algorithm searches successively smaller arrays: The size of a given array is approximately one-half the size of the array previously searched.

At each division, the algorithm makes a comparison. How many comparisons does the algorithm make when it searches an array of n items? The exact answer depends, of course, on where the sought-for item resides in the array. However, you can compute the maximum number of comparisons that a binary search requires—that is, the worst case. The number of comparisons is equal to the number of times that the algorithm divides the array in half. Suppose that $n = 2^k$ for some k. The search requires the following steps:

1. Inspect the middle item of an array of size n.
2. Inspect the middle item of an array of size $n/2$.
3. Inspect the middle item of an array of size $n/2^2$, and so on.

To inspect the middle item of an array, you must first divide the array in half. If you divide an array of n items in half, then divide one of those halves in half, and continue dividing halves until only one item remains, you will have performed k divisions. This is true because $n/2^k = 1$. (Remember, we assumed that $n = 2^k$.) In the worst case, the algorithm performs k divisions and, therefore, k comparisons. Because $n = 2^k$,

$k = \log_2 n$

Thus, the algorithm is $O(\log_2 n)$ in the worst case when $n = 2^k$.

What if n is not a power of 2? You can easily find the smallest k such that

$$2^{k-1} < n < 2^k$$

For example, if n is 30, then $k = 5$, because $2^4 = 16 < 30 < 32 = 2^5$. The algorithm still requires at most k divisions to obtain a subarray with one item. Now it follows that

$$k - 1 < \log_2 n < k$$

$$k < 1 + \log_2 n < k + 1$$

$k = 1 + \log_2 n$ rounded down

A binary search is $O(\log_2 n)$ in the worst case

Thus, the algorithm is still $O(\log_2 n)$ in the worst case when $n = 2^k$. In general, the algorithm is $O(\log_2 n)$ in the worst case for any n.

Is a binary search better than a sequential search? Much better! For example $\log_2 1{,}000{,}000 = 19$, so a sequential search of one million sorted items can require one million comparisons, but a binary search of the same items will require at most 20 comparisons. For large arrays, the binary search has an enormous advantage over a sequential search.

Realize, however, that maintaining the array in sorted order requires an overhead cost, which can be substantial. The next section examines the cost of sorting an array.

10.2 Sorting Algorithms and Their Efficiency

Sorting is a process that organizes a collection of data into either ascending[4] or descending order. The need for sorting arises in many situations. You may simply want to sort a collection of data before including it in a report. Often, however, you must perform a sort as an initialization step for certain algorithms. For example, searching for data is one of the most common tasks performed by computers. When the collection of data to be searched is large, an efficient method for searching—such as the binary search algorithm—is desirable. However, the binary search algorithm requires that the data be sorted. Thus, sorting the data is a step that must precede a binary search on a collection of data that is not already sorted. Good sorting algorithms, therefore, are quite valuable.

The sorts in this chapter are internal sorts

You can organize sorting algorithms into two categories. An **internal sort** requires that the collection of data fit entirely in the computer's main memory. The algorithms in this chapter are internal sorting algorithms. You use an

4. To allow for duplicate data items, "ascending" is used here to mean nondecreasing and "descending" to mean nonincreasing.

external sort when the collection of data will not fit in the computer's main memory all at once but must reside in secondary storage, such as on a disk.

The data items to be sorted might be integers, character strings, or even objects. It is easy to imagine the results of sorting a collection of integers or character strings, but consider a collection of objects. If each object contains only one data field, sorting the objects is really no different than sorting a collection of integers. However, when each object contains several data fields, you must know which data field determines the order of the entire object within the collection of data. This data field is called the **sort key.** For example, if the objects represent people, you might want to sort on their names, their ages, or their zip codes. Regardless of your choice of sort key, the sorting algorithm orders entire objects based on only one data field, the sort key.

For simplicity, this chapter assumes that the data items are instances of a class that has implemented the `Comparable` interface. The `Comparable` interface method `compareTo` returns either a negative integer, zero, or a positive integer based upon whether the sort key is less than, equal to, or greater than the sort key of the specified object. All algorithms in this chapter sort the data into ascending order. Modifying these algorithms to sort data into descending order is simple. Finally, each example assumes that the data resides in an array.

Selection Sort

Imagine some data that you can examine all at once. To sort it, you could select the largest item and put it in its place, select the next largest and put it in its place, and so on. For a card player, this process is analogous to looking at an entire hand of cards and ordering it by selecting cards one at a time in their proper order. The **selection sort** formalizes these intuitive notions. To sort an array into ascending order, you first search it for the largest item. Because you want the largest item to be in the last position of the array, you swap the last item with the largest item, even if these items happen to be identical. Now, ignoring the last—and largest—item of the array, you search the rest of the array for its largest item and swap it with its last item, which is the next-to-last item in the original array. You continue until you have selected and swapped $n - 1$ of the n items in the array. The remaining item, which is now in the first position of the array, is in its proper order, so it is not considered further.

Select the largest item

Figure 10-4 provides an example of a selection sort. Beginning with five integers, you select the largest—37—and swap it with the last integer—13. (As the items in this figure are ordered, they appear in boldface. This convention will be used throughout this chapter.) Next you select the largest integer—29—from among the first four integers in the array and swap it with the next-to-last integer in the array—13. Notice that the next selection—14—is already in its proper position, but the algorithm ignores this fact and performs a swap of 14 with itself. It is more efficient in general to occasionally perform an unnecessary swap than it is to continually ask whether the swap is necessary.

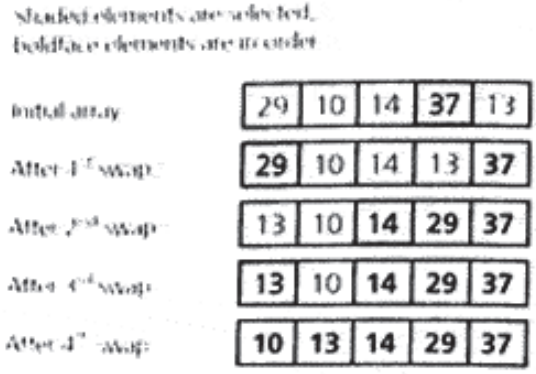


FIGURE 10-4

A selection sort of an array of five integers

Finally, you select the 13 and swap it with the item in the second position of the array—10. The array is now sorted into ascending order.

A Java method that performs a selection sort on an array called `theArray` with `n` items follows:

```
public static <T extends Comparable<? super T>>
      void selectionSort(T[] theArray, int n) {
// ------------------------------------------------
// Sorts the items in an array into ascending order.
// Precondition: theArray is an array of n items.
// Postcondition: theArray is sorted into
// ascending order.
// Calls: indexOfLargest.
// ------------------------------------------------
 // last = index of the last item in the subarray of
 //        items yet to be sorted
 // largest = index of the largest item found

 for (int last = n-1; last >= 1; last--) {
   // Invariant: theArray[last+1..n-1] is sorted
   // and > theArray[0..last]

   // select largest item in theArray[0..last]
   int largest = indexOfLargest(theArray, last+1);

   // swap largest item theArray[largest] with
   // theArray[last]
   T temp = theArray[largest];
```

```
    theArray[largest] = theArray[last];
    theArray[last] = temp;
  } // end for
} // end selectionSort
```

The `selectionSort` method calls the following method:

```
private static <T extends Comparable<? super T>>
        int indexOfLargest(T[] theArray, int size) {
  ------------------------------------------------
  Finds the largest item in an array.
  Precondition: theArray is an array of size items;
  size >= 1.
  Postcondition: Returns the index of the largest
  item in the array.
  ------------------------------------------------
 int indexSoFar = 0; // index of largest item found so far

   Invariant: theArray[indexSoFar]>=theArray[0..currIndex-1]
 for (int currIndex = 1; currIndex < size; ++currIndex) {
   if (theArray[currIndex].compareTo(theArray[indexSoFar])>0) {
     indexSoFar = currIndex;
   } // end if
 } // end for

 return indexSoFar;  // index of largest item
} // end indexOfLargest
```

Analysis. As you can see from the previous algorithm, sorting in general compares, exchanges, or moves items. Depending on the programming language and implementation used for storing the data, the cost associated with each of these operations varies. For example, in Java, an array stores references to objects, not the objects themselves. Thus, moving or exchanging data is not expensive in Java, because only the references are moved or exchanged, rather than entire objects. Other languages, however, such as C++, could use an implementation in which the actual objects are stored in an array. In this case, moving and exchanging data becomes much more expensive since entire objects are actually moved around memory.

The comparison operation is typically more involved, since actual data values must be compared. In Java, one way to compare data values is through the implementation of the `java.util.Comparable` interface, and in particular, the `compareTo` method. In Java, the use of the `Comparable` interface provides what is referred to as the ***natural ordering*** for a class. Alternatively, you can create a class that implements the interface `java.util.Comparator`. It provides a method `compare` which imposes a ***total ordering*** on some collection of

objects. These two approaches to implementing the element comparison are discussed later in this chapter. In either case, the comparison itself is usually based upon a portion of the object, that is, the sort key.

As a first step in analyzing sorting algorithms, you should count the move, exchange, and compare operations. In Java, the comparison operation is usually the most expensive and is thus often the only operation analyzed. However, since other programming languages may incur more expense in moving or exchanging data, we provide some analysis of these operations here. Generally, move, exchange, and compare operations are more expensive than the ones that control loops or manipulate array indexes, particularly when the data to be sorted is more complex than integers or characters. Thus, our approach ignores these less expensive operations. You should convince yourself that ignoring such operations does not affect our final result. (See Exercise 7.)

Clearly, the `for` loop in the method `selectionSort` executes $n - 1$ times. Thus, `selectionSort` calls the method `indexOfLargest` $n - 1$ times. Each call to `indexOfLargest` causes its loop to execute `last` times (that is, `size` – 1 times when `size` is `last` + 1). Thus, the $n - 1$ calls to `indexOfLargest`, for values of `last` that range from $n - 1$ down to 1, cause the loop in `indexOfLargest` to execute a total of

$$(n-1) + (n-2) + \cdots + 1 = n * (n-1)/2$$

times. Because each execution of `indexOfLargest`'s loop performs one comparison, the calls to `indexOfLargest` require

$$n * (n-1)/2$$

comparisons.

At the end of the `for` loop in `selectionSort`, an exchange is performed between elements `theArray[largest]` and `theArray[last]`. Each exchange requires three assignments, or

$$3 * (n-1)$$

data moves.

Together, a selection sort of n items requires

$$n * (n-1)/2 + 3 * (n-1)$$

$$= n^2/2 + 5 * n/2 - 3$$

Selection sort is $O(n^2)$

major operations. By applying the properties of growth-rate functions (see page 508), you can ignore low-order terms to get $O(n^2/2)$ and then ignore the multiplier $1/2$ to get $O(n^2)$. Thus, selection sort is $O(n^2)$.

Although a selection sort does not depend on the initial arrangement of the data, which is an advantage of this algorithm, it is appropriate only for small n because $O(n^2)$ grows rapidly. While the algorithm requires $O(n^2)$ comparisons, it requires only $O(n)$ data moves. A selection sort could be a good choice over other methods when data moves are costly but comparisons are not. As we mentioned earlier, there really is no such thing as an expensive data move in a normal sorting situation in Java, because it would surely be references, not entire objects, that are copied in the process of performing the move.

Bubble Sort

The next sorting algorithm is one that you may have seen already. That is precisely why it is analyzed here, because it is not a particularly good algorithm. The **bubble sort** compares adjacent items and exchanges them if they are out of order. This sort usually requires several passes over the data. During the first pass, you compare the first two items in the array. If they are out of order, you exchange them. You then compare the items in the next pair—that is, in positions 2 and 3 of the array. If they are out of order, you exchange them. You proceed, comparing and exchanging items two at a time until you reach the end of the array.

Figure 10-5a illustrates the first pass of a bubble sort of an array of five integers. You compare the items in the first pair—29 and 10—and exchange them because they are out of order. Next you consider the second pair—29 and 14—and exchange these items because they are out of order. The items in the third pair—29 and 37—are in order, and so you do not exchange them. Finally, you exchange the items in the last pair—37 and 13.

When you order successive pairs of items, the largest item bubbles to the top (end) of the array

Although the array is not sorted after the first pass, the largest item has "bubbled" to its proper position at the end of the array. During the second pass of the bubble sort, you return to the beginning of the array and consider pairs of items in exactly the same manner as the first pass. You do not, however, include the last—and largest—item of the array. That is, the second pass considers the first $n - 1$ items of the array. After the second pass, the second largest item in the array will be in its proper place in the next-to-last position of the array, as Figure 10-5b illustrates. Now, ignoring the last two items, which are in order, you continue with subsequent passes until the array is sorted.

Bubble sort usually requires several passes through the array

Although a bubble sort requires at most $n - 1$ passes to sort the array, fewer passes might be possible to sort a particular array. Thus, you could terminate the process if no exchanges occur during any pass. The following Java method `bubbleSort` uses a flag to signal when an exchange occurs during a particular pass.

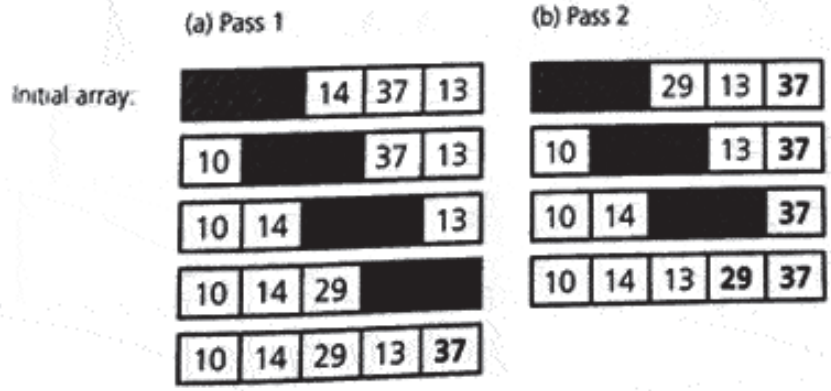


FIGURE 10-5

The first two passes of a bubble sort of an array of five integers: (a) pass 1; (b) pass 2

```
public static <T extends Comparable<? super T>>
       void bubbleSort(T[] theArray, int n) {
// ------------------------------------------------
// Sorts the items in an array into ascending order.
// Precondition: theArray is an array of n items.
// Postcondition: theArray is sorted into ascending
// order.
// ------------------------------------------------
  boolean sorted = false;  // false when swaps occur

  for (int pass = 1; (pass < n) && !sorted; ++pass) {
    // Invariant: theArray[n+1-pass..n-1] is sorted
    //            and > theArray[0..n-pass]
    sorted = true;  // assume sorted
    for (int index = 0; index < n-pass; ++index) {
      // Invariant: theArray[0..index-1] <= theArray[index]
      int nextIndex = index + 1;
      if (theArray[index].compareTo(theArray[nextIndex]) > 0) {
        // exchange items
        T temp = theArray[index];
        theArray[index] = theArray[nextIndex];
        theArray[nextIndex] = temp;
        sorted = false;  // signal exchange
      }  // end if
    }  // end for

    // Assertion: theArray[0..n-pass-1] < theArray[n-pass]
  }  // end for
}  // end bubbleSort
```

Analysis. As was noted earlier, the bubble sort requires at most $n - 1$ passes through the array. Pass 1 requires $n - 1$ comparisons and at most $n - 1$ exchanges; pass 2 requires $n - 2$ comparisons and at most $n - 2$ exchanges. In general, pass i requires $n - i$ comparisons and at most $n - i$ exchanges. Therefore, in the worst case, a bubble sort will require a total of

$$(n - 1) + (n - 2) + \cdots + 1 = n * (n - 1)/2$$

comparisons and the same number of exchanges. Recall that each exchange requires three data moves. Thus, altogether there are

$$2 * n * (n - 1) = 2 * n^2 - 2 * n$$

major operations in the worst case. Therefore, the bubble sort algorithm is $O(n^2)$ in the worst case.

Bubble sort. Worst case: $O(n^2)$; best case: $O(n)$

The best case occurs when the original data is already sorted: *bubbleSort* uses one pass, during which $n - 1$ comparisons and no exchanges occur. Thus, the bubble sort is $O(n)$ in the best case.

Insertion Sort

Imagine once again arranging a hand of cards, but now you pick up one card at a time and insert it into its proper position; in this case you are performing an **insertion sort.** Chapter 5 introduced the insertion sort algorithm in the context of a linked list: You can create a sorted linked list from a file of unsorted integers, for example, by repeatedly calling a method that inserts an integer into its proper sorted order in a linked list.

You can use the insertion sort strategy to sort items that reside in an array. This version of the insertion sort partitions the array into two regions: sorted and unsorted, as Figure 10-6 depicts. Initially, the entire array is the unsorted region, just as the cards dealt to you sit in an unsorted pile on the table. At each step, the insertion sort takes the first item of the unsorted region and places it into its correct position in the sorted region. This step is analogous to taking a card from the table and inserting it into its proper position in your hand. The first step, however, is trivial: Moving `theArray[0]` from the unsorted region to the sorted region really does not require moving data. Therefore, you can omit this first step by considering the initial sorted region to be `theArray[0]` and the initial unsorted region to be `theArray[1..n-1]`. The fact that the items in the sorted region are sorted among themselves is an invariant of the algorithm. Because at each step the size of the sorted region grows by 1 and the size of the unsorted region shrinks by 1, the entire array will be sorted when the algorithm terminates.

Take each item from the unsorted region and insert it into its correct order in the sorted region

Figure 10-7 illustrates an insertion sort of an array of five integers. Initially, the sorted region is `theArray[0]`, which is 29, and the unsorted region is the rest of the array. You take the first item in the unsorted region—the 10—and insert it into its proper position in the sorted region. This insertion requires you to shift array items to make room for the inserted item. You then take the first item in the new unsorted region—the 14—and insert it into its proper position in the sorted region, and so on.

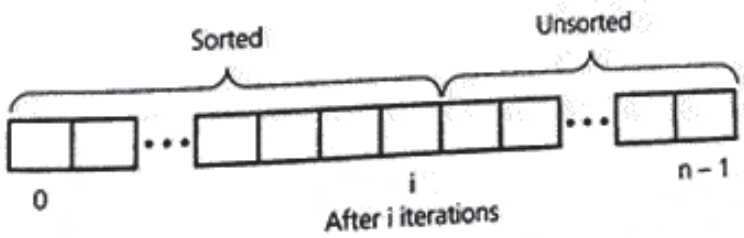


FIGURE 10-6

An insertion sort partitions the array into two regions

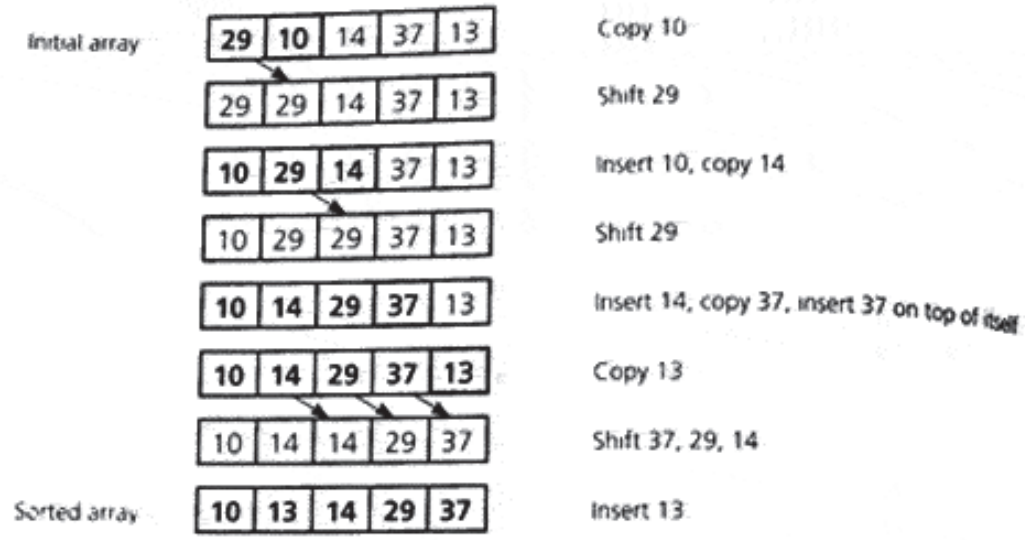


FIGURE 10-7

An insertion sort of an array of five integers

A Java method that performs an insertion sort on an array of n items follows:

```
public static <T extends Comparable<? super T>>
       void insertionSort(T[] theArray, int n) {
// ------------------------------------------------
// Sorts the items in an array into ascending order.
// Precondition: theArray is an array of n items.
// Postcondition: theArray is sorted into ascending
// order.
// ------------------------------------------------
  // unsorted = first index of the unsorted region,
  // loc = index of insertion in the sorted region,
  // nextItem = next item in the unsorted region

  // initially, sorted region is theArray[0],
  //          unsorted region is theArray[1..n-1];
  // in general, sorted region is theArray[0..unsorted-1],
  //           unsorted region is theArray[unsorted..n-1]

  for (int unsorted = 1; unsorted < n; ++unsorted) {
    // Invariant: theArray[0..unsorted-1] is sorted

    // find the right position (loc) in
    // theArray[0..unsorted] for theArray[unsorted],
    // which is the first item in the unsorted
    // region; shift, if necessary, to make room
    T nextItem = theArray[unsorted];
```

```
    int loc = unsorted;

    while ((loc > 0) &&
           (theArray[loc-1].compareTo(nextItem) > 0)) {
      // shift theArray[loc-1] to the right
      theArray[loc] = theArray[loc-1];
      loc--;
    }  // end while
    // Assertion: theArray[loc] is where nextItem belongs
    // insert nextItem into sorted region
    theArray[loc] = nextItem;
  }  // end for
}  // end insertionSort
```

Analysis. The outer *for* loop in the method *insertionSort* executes $n - 1$ times. This loop contains an inner *for* loop that executes at most *unsorted* times for values of *unsorted* that range from 1 to $n - 1$. Thus, in the worst case, the algorithm's comparison occurs

$$1 + 2 + \cdots + (n - 1) = n * (n - 1)/2$$

times. In addition, the inner loop moves data items at most the same number of times.

The outer loop moves data items twice per iteration, or $2 * (n - 1)$ times. Together, there are

$$n * (n - 1) + 2 * (n - 1) = n^2 + n - 2$$

major operations in the worst case.

Therefore, the insertion sort algorithm is $O(n^2)$ in the worst case. For small arrays—say, fewer than 25 items—the simplicity of the insertion sort makes it an appropriate choice. For large arrays, however, an insertion sort can be prohibitively inefficient.

Insertion sort is $O(n^2)$ in the worst case

Mergesort

Two important divide-and-conquer sorting algorithms, mergesort and quicksort, have elegant recursive formulations and are highly efficient. The presentations here are in the context of sorting arrays, but note that mergesort generalizes to external files. It will be convenient to express the algorithms in terms of the array *theArray[first..last]*.

Divide and conquer

Mergesort is a recursive sorting algorithm that always gives the same performance, regardless of the initial order of the array items. Suppose that you divide the array into halves, sort each half, and then merge the sorted halves into one sorted array, as Figure 10-8 illustrates. In the figure, the halves <1, 4, 8> and <2, 3> are merged to form the array <1, 2, 3, 4, 8>. This merge step compares an item in one half of the array with an item in the other half and moves the smaller item to a temporary array. This process continues until there are no more items to consider in one half. At that time, you simply

Halve the array, recursively sort its halves, and then merge the halves

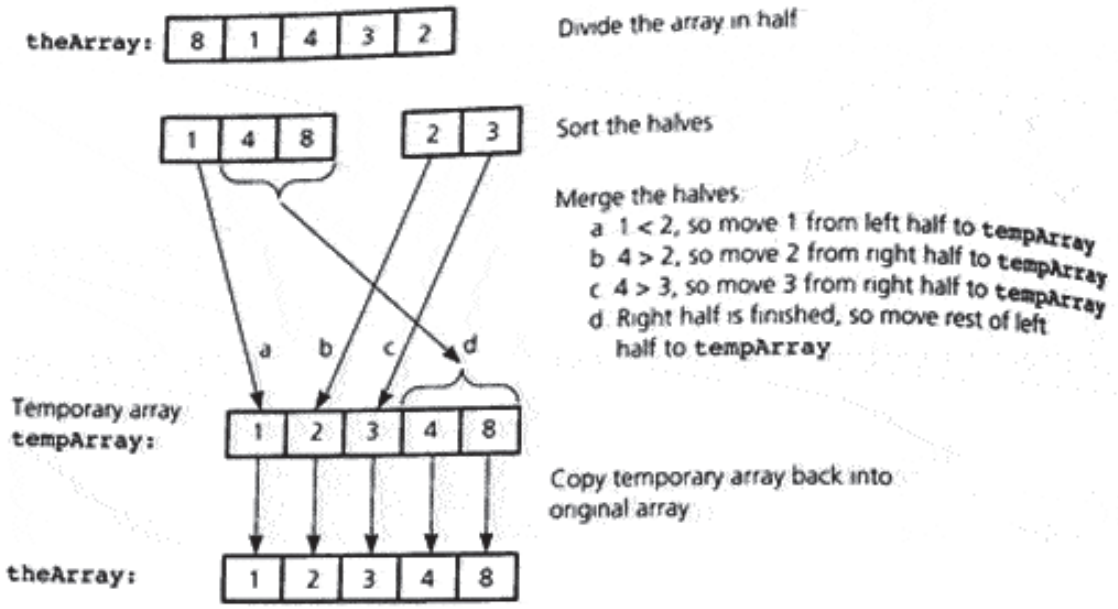


FIGURE 10-8

A mergesort with an auxiliary temporary array

move the remaining items to the temporary array. Finally, you copy the temporary array back into the original array.

Although the merge step of mergesort produces a sorted array, how do you sort the array halves prior to the merge step? Mergesort sorts the array halves by using mergesort—that is, by calling itself recursively. Thus, the pseudocode for mergesort is

```
+mergesort(inout theArray:ItemArray,
           in first:integer, in last:integer)
// Sorts theArray[first..last] by
//   1. sorting the first half of the array
//   2. sorting the second half of the array
//   3. merging the two sorted halves

 if (first < last) {
   mid = (first + last)/2   // get midpoint
   // sort theArray[first..mid]
   mergesort(theArray, first, mid)
   // sort theArray[mid+1..last]
   mergesort(theArray, mid + 1, last)
   // merge sorted halves theArray[first..mid]
   // and theArray[mid+1..last]
   merge(theArray, first, mid, last)
 } // end if
 // if first >= last, there is nothing to do
```

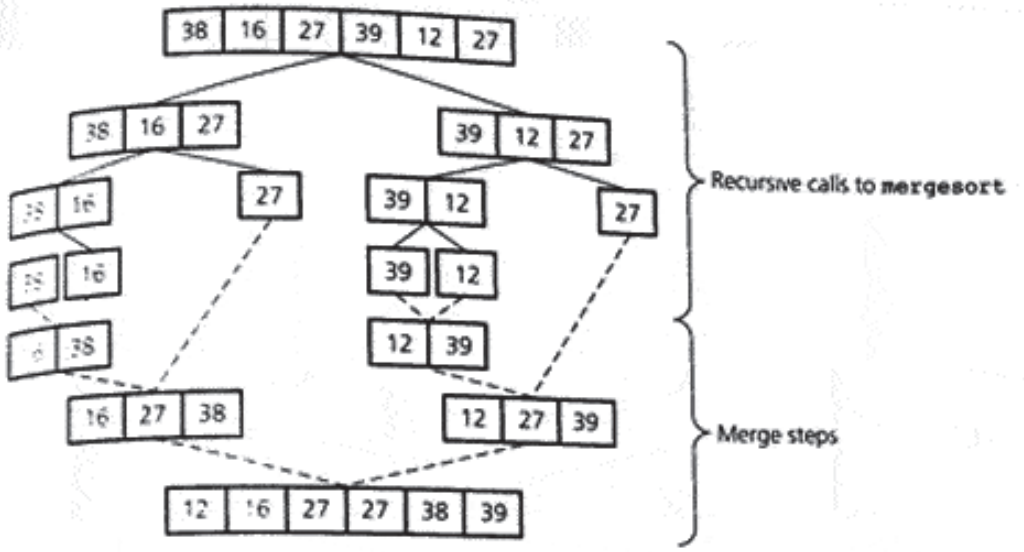


FIGURE 10-9

A mergesort of an array of six integers

Clearly, most of the effort in the mergesort algorithm is in the merge step, but does this algorithm actually sort? The recursive calls continue dividing the array into pieces until each piece contains only one item; obviously an array of one item is sorted. The algorithm then merges these small pieces into larger sorted pieces until one sorted array results. Figure 10-9 illustrates both the recursive calls and the merge steps in a mergesort of an array of six integers.

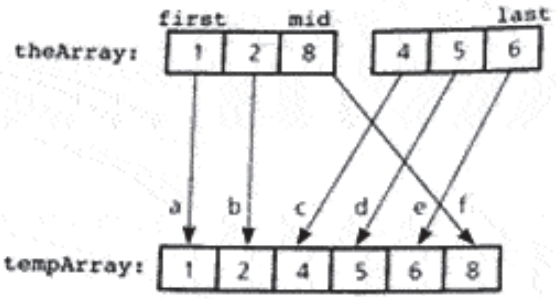
The following Java methods implement the mergesort algorithm. To sort an array *theArray* of *n* items, you would invoke the method *mergesort* by writing *mergesort(theArray)*. Notice that this method creates a reusable temporary array that can be used later on for the merge. It then calls the recursive function *mergesort(theArray, tempArray, 0, theArray.length-1)* with this temporary array to actually complete the sort.

```
public static<T extends Comparable<? super T>>
      void mergesort(T[ ] theArray) {
// Declare temporary array used for merge, must do
// unchecked cast from Comparable<?>[] to T[]

  T[] tempArray = (T[])new Comparable<?>[theArray.length];
  mergesort(theArray, tempArray, 0, theArray.length - 1 );
} // end mergesort

private static<T extends Comparable<? super T>>
       void merge(T[] theArray, T[] tempArray,
                  int first, int mid, int last) {
```

```
// ---------------------------------------------------------
// Merges two sorted array segments theArray[first..mid] and
// theArray[mid+1..last] into one sorted array.
// Precondition: first <= mid <= last. The subarrays
// theArray[first..mid] and theArray[mid+1..last] are
// each sorted in increasing order.
// Postcondition: theArray[first..last] is sorted.
// Implementation note: This method merges the two
// subarrays into a temporary array and copies the result
// into the original array theArray.
// ---------------------------------------------------------

  // initialize the local indexes to indicate the subarrays
  int first1 = first;     // beginning of first subarray
  int last1  = mid;       // end of first subarray
  int first2 = mid + 1;   // beginning of second subarray
  int last2  = last;      // end of second subarray
  // while both subarrays are not empty, copy the
  // smaller item into the temporary array
  int index = first1;     // next available location in
                          // tempArray

 while ((first1 <= last1) && (first2 <= last2)) {
   // Invariant: tempArray[first1..index-1] is in order
   if (theArray[first1].compareTo(theArray[first2])<0) {
     tempArray[index] = theArray[first1];
     first1++;
   }
   else {
     tempArray[index] = theArray[first2];
     first2++;
   }  // end if
   index++;
 }  // end while

 // finish off the nonempty subarray

 // finish off the first subarray, if necessary
 while (first1 <= last1) {
   // Invariant: tempArray[first1..index-1] is in order
   tempArray[index] = theArray[first1];
   first1++;
   index++;
 }  // end while
```

```
    // finish off the second subarray, if necessary
    while (first2 <= last2) {
      // Invariant: tempArray[first1..index-1] is in order
      tempArray[index] = theArray[first2];
      first2++;
      index++;
    } // end while

    // copy the result back into the original array
    for (index = first; index <= last; ++index) {
      theArray[index] = tempArray[index];
    } // end for
  } // end merge

  public static <T extends Comparable<? super T>>
        void mergesort(T[] theArray, T[] tempArray
                       int first, int last) {
  // -------------------------------------------------------
  // Sorts the items in an array into ascending order.
  // Precondition: theArray[first..last] is an array.
  // Postcondition: theArray[first..last] is sorted in
  // ascending order.
  // Calls: merge.
  // -------------------------------------------------------
    if (first < last) {
      // sort each half
      int mid = (first + last)/2;   // index of midpoint
      // sort left half theArray[first..mid]
      mergesort(theArray, tempArray, first, mid);
      // sort right half theArray[mid+1..last]
      mergesort(theArray, tempArray, mid+1, last);

      // merge the two halves
      merge(theArray, tempArray, first, mid, last);
    } // end if
  } // end mergesort
```

Analysis. Because the merge step of the algorithm requires the most effort, let's begin the analysis there. Each merge step merges *theArray[first..mid]* and *theArray[mid+1..last]*. Figure 10-10 provides an example of a merge step that requires the maximum number of comparisons. If the total number of items in the two array segments to be merged is n, then merging the segments requires at most $n - 1$ comparisons. (For example, in Figure 10-10 there are six items in the segments and five comparisons.) In addition, there are n moves from the original array to the temporary array, and n moves from the temporary array back to the original array. Thus, each merge step requires $3 * n - 1$ major operations.



Merge the halves
a 1 < 4, so move 1 from `theArray[first..mid]` to `tempArray`
b 2 < 4, so move 2 from `theArray[first..mid]` to `tempArray`
c 8 > 4, so move 4 from `theArray[mid+1..last]` to `tempArray`
d 8 > 5, so move 5 from `theArray[mid+1..last]` to `tempArray`
e 8 > 6, so move 6 from `theArray[mid+1..last]` to `tempArray`
f `theArray[mid+1..last]` is finished, so move 8 to `tempArray`

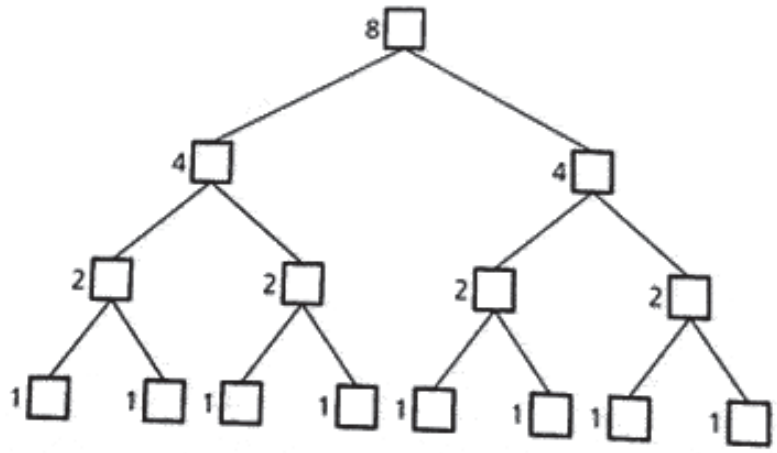
FIGURE 10-10

A worst-case instance of the merge step in ***mergesort***

Each call to ***mergesort*** recursively calls itself twice. As Figure 10-11 illustrates, if the original call to ***mergesort*** is at level 0, two calls to ***mergesort*** occur at level 1 of the recursion. Each of these calls then calls ***mergesort*** twice, so four calls to ***mergesort*** occur at level 2 of the recursion, and so on. How many levels of recursion are there? We can count them as follows.

Each call to ***mergesort*** halves the array. Halving the array the first time produces two pieces. The next recursive calls to ***mergesort*** halve each of these two pieces to produce four pieces of the original array; the next recursive calls halve each of these four pieces to produce eight pieces, and so on. The recursive calls continue until the array pieces each contain one item—that is, until there are n pieces, where n is the number of items in the original array. If n is a power of 2 ($n = 2^k$), then the recursion goes $k = \log_2 n$ levels deep. For example, in Figure 10-11, there are three levels of recursive calls to ***mergesort*** because the original array contains eight items, and $8 = 2^3$. If n is not a power of 2, there are $1 + \log_2 n$ (rounded down) levels of recursive calls to ***mergesort***.

The original call to ***mergesort*** (at level 0) calls ***merge*** once. When called, ***merge*** merges all n items and requires $3 * n - 1$ operations, as was shown earlier. At level 1 of the recursion, two calls to ***mergesort***, and hence to ***merge***,



Level 0: `mergesort` 8 items

Level 1: 2 calls to `mergesort` with 4 items each

Level 2: 4 calls to `mergesort` with 2 items each

Level 3: 8 calls to `mergesort` with 1 item each

FIGURE 10-11

Levels of recursive calls to ***mergesort***, given an array of eight items

occur. Each of these two calls to `merge` merges $n/2$ items and requires $3 * (n/2) - 1$ operations. Together these two calls to `merge` require $2 * 3 * (n/2) - 1$), or $3 * n - 2$ operations. At level m of the recursion, 2^m calls to `merge` occur; each of these calls merges $n/2^m$ items and so requires $3 * (n/2^m) - 1$ operations. Together the 2^m calls to `merge` require $3 * n - 2^m$ operations. Thus, each level of the recursion requires $O(n)$ operations. Because there are either $\log_2 n$ or $1 + \log_2 n$ levels, `mergesort` is $O(n * \log_2 n)$ in both the worst and average cases. You should look at Figure 10-3 again to convince yourself that $O(n * \log_2 n)$ is significantly faster than $O(n^2)$.

Mergesort is $O(n * \log_2 n)$

Although mergesort is an extremely efficient algorithm with respect to time, it does have one drawback: To perform the step

```
merge sorted halves theArray[first..mid]
   and theArray[mid+1..last]
```

the algorithm requires an auxiliary array whose size equals the size of the original array. In Java, this auxiliary array is simply an array of references and hence has little impact. Other programming languages, however, such as C++, actually store the data items in the array. With such languages, this requirement might not be acceptable in situations where storage is limited.

Mergesort requires a second array as large as the original array

Quicksort

Consider the first two steps of the solution to the problem of finding the k^{th} smallest item of the array `theArray[first..last]` that was discussed in Chapter 3:

Another divide-and-conquer algorithm

```
Choose a pivot item p from theArray[first..last]
Partition the items of theArray[first..last] about p
```

Recall that this partition, which is pictured again in Figure 10-12, has the property that all items in S_1 = `theArray[first..pivotIndex-1]` are less than the pivot p, and all items in S_2 = `theArray[pivotIndex+1..last]` are greater than or equal to p. Although this property does not imply that the array is sorted, it does imply an extremely useful fact: The items in positions `first` through `pivotIndex` – 1 remain in positions `first` through `pivotIndex` – 1

Quicksort partitions an array into items that are less than the pivot and those that are greater than or equal to the pivot

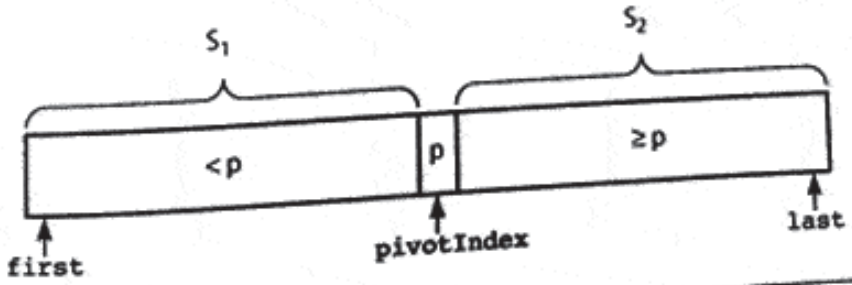


FIGURE 10-12

A partition about a pivot

Partitioning places the pivot in its correct position within the array

when the array is properly sorted, although their positions relative to one another may change. Similarly, the items in positions `pivotIndex` + 1 through `last` will remain in positions `pivotIndex` + 1 through `last` when the array is sorted, although their relative positions may change. Finally, the pivot item remains in its position in the final, sorted array.

The partition induces relationships among the array items that are the ingredients of a recursive solution. Arranging the array items around the pivot p generates two smaller sorting problems—sort the left section of the array (S_1), and sort the right section of the array (S_2). The relationships between the pivot and the array items imply that once you solve the left and right sorting problems, you will have solved the original sorting problem. That is, partitioning the array before making the recursive calls places the pivot in its correct position and ensures that when the smaller array segments are sorted, their items will be in the proper relation to the rest of the array. Also, the quicksort algorithm will eventually terminate: The left and right sorting problems are indeed smaller problems and are each closer than the original sorting problem to the base case—which is an array containing one item—because the pivot is not part of either S_1 or S_2.

The pseudocode for the quicksort algorithm follows:

```
+quicksort(inout theArray:ItemArray,
           in first:integer, in last:integer)
// Sorts theArray[first..last].

  if (first < last) {
    Choose a pivot item p from theArray[first..last]
    Partition the items of theArray[first..last] about p
    // the partition is theArray[first..pivotIndex..last]

    // sort S1
    quicksort(theArray, first, pivotIndex-1)
    // sort S2
    quicksort(theArray, pivotIndex+1, last)
  }  // end if
// if first >= last, there is nothing to do
```

It is worth contrasting `quicksort` with the pseudocode method given for the k^{th} smallest integer problem in Chapter 3:

```
+kSmall(in k:integer, in theArray:ItemArray,
        in first:integer, in last:integer):ItemType
// Returns the k^th smallest value in theArray[first..last].

 Choose a pivot item p from theArray[first..last]
 Partition the items of theArray[first..last] about p
 if (k < pivotIndex - first + 1) {
   return kSmall(k, theArray, first, pivotIndex-1)
 }
```

```
else if (k == pivotIndex - first + 1) {
  return p
}
else {
  return kSmall(k-(pivotIndex-first+1),
                theArray, pivotIndex+1, last)
}  // end if
```

Note that *kSmall* is called recursively only on the section of the array that contains the desired item, and it is not called at all if the desired item is the pivot. On the other hand, *quicksort* is called recursively on both unsorted sections of the array. Figure 10-13 illustrates this difference.

Difference between *kSmall* and *quicksort*

Using an invariant to develop a partition algorithm. Now consider the partition method that both *kSmall* and *quicksort* must call. Partitioning an array section about a pivot item is actually the most difficult part of these two problems.

The partition method will receive an array segment *theArray[first..last]* as an argument. The method must arrange the items of the array segment into two regions: S_1, the set of items less than the pivot, and S_2, the set of items greater than or equal to the pivot. The method arranges the array so that S_1 is *theArray[first..pivotIndex-1]* and S_2 is *theArray[pivotIndex+1..last]*, as you saw in Figure 10-12.

What pivot should you use? If the items in the array are arranged randomly, you can choose a pivot at random. For example, you can choose *theArray[first]* as the pivot. (The choice of pivot will be discussed in more detail later.) While you are

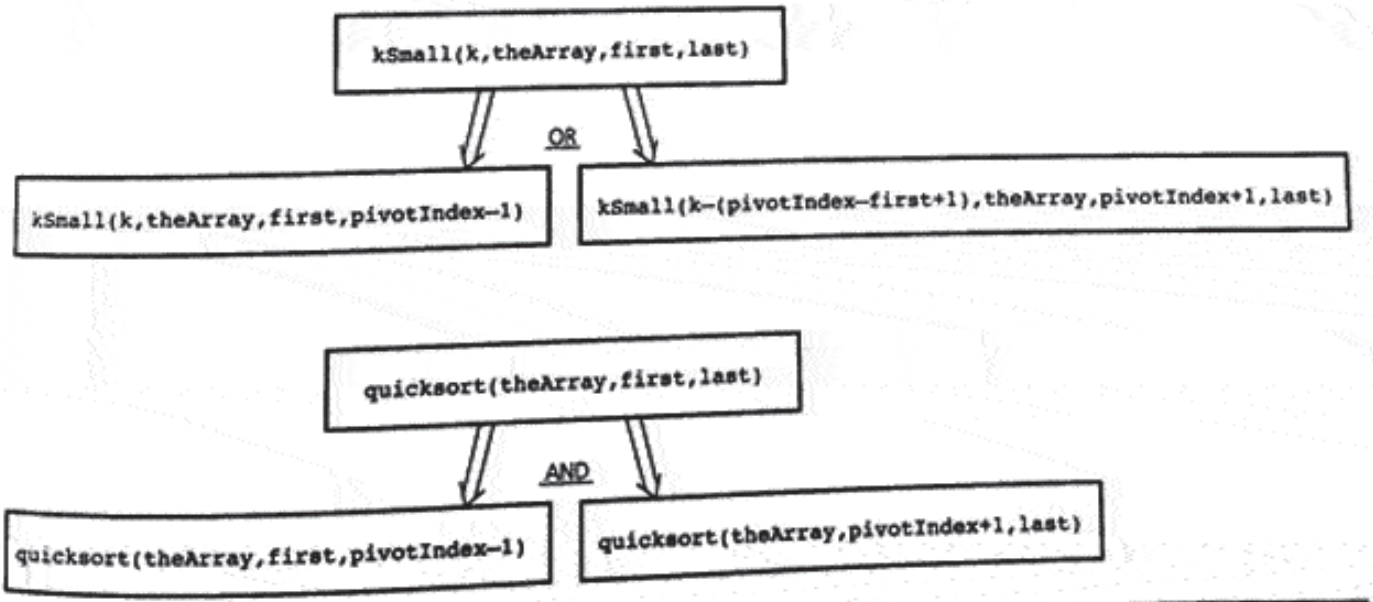


FIGURE 10-13

kSmall* versus *quicksort

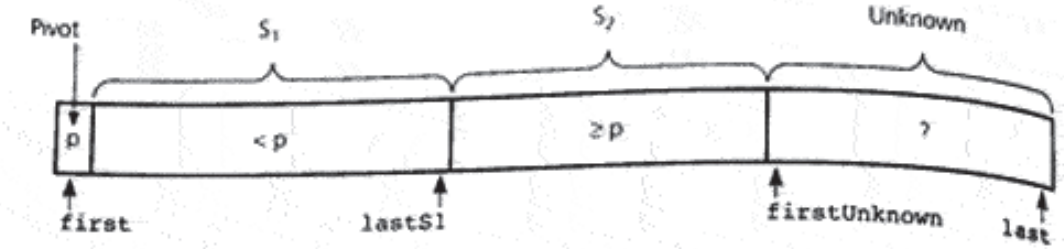


FIGURE 10-14

Invariant for the partition algorithm

Place your choice of pivot in **`theArray[first]`** before partitioning

developing the partition, it is convenient to place the pivot in the `theArray[first]` position, regardless of which pivot you choose.

The items that await placement into either S_1 or S_2 are in another region of the array, called the unknown region. Thus, you should view the array as shown in Figure 10-14. The array indexes `first`, $lastS_1$, `firstUnknown`, and `last` divide the array as just described. The relationships between the pivot and the items in the unknown region—which is `theArray[firstUnknown..last]`—are, simply, unknown!

Throughout the entire partitioning process, the following is true:

Invariant for the partition algorithm

> *The items in the region S_1 are all less than the pivot, and those in S_2 are all greater than or equal to the pivot.*

This statement is the invariant for the partition algorithm. For the invariant to be true at the start of the partition algorithm, the array's indexes must be initialized as follows so that the unknown region spans all of the array segment to be partitioned except the pivot:

Initially, all items except the pivot **`theArray[first]`** constitute the unknown region

```
lastS1 = first
firstUnknown = first + 1
```

Figure 10-15 shows the initial status of the array.

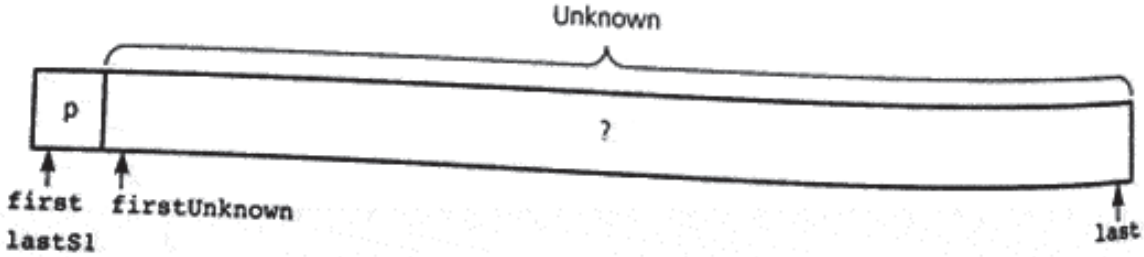


FIGURE 10-15

Initial state of the array

At each step of the partition algorithm, you examine one item of the unknown region, determine in which of the two regions, S_1 or S_2, it belongs, and place it there. Thus, the size of the unknown region decreases by 1 at each step. The algorithm terminates when the size of the unknown region reaches 0—that is, when `firstUnknown > last`.

The following pseudocode describes the partitioning algorithm:

The partition algorithm

```
partition(inout theArray:ItemArray,
          in first:integer, in last:integer)
Returns the index of the pivot element after
partitioning theArray[first..last].

  initialize
choose the pivot and swap it with theArray[first]
p = theArray[first]          // p is the pivot
lastS1 = first               // set S1 and S2 to empty
firstUnknown = first + 1     // set unknown region
                             // to theArray[first+1..last]
  determine the regions S1 and S2
while (firstUnknown <= last) {
   consider the placement of the "leftmost"
   item in the unknown region
  if (theArray[firstUnknown] < p) {
    Move theArray[firstUnknown] into S1
  }
  else {
    Move theArray[firstUnknown] into S2
  }  // end if
}  // end while
// place pivot in proper position between
// S1 and S2, and mark its new location
Swap theArray[first] with theArray[lastS1]
return lastS1 // the index of the pivot element
```

The algorithm is straightforward enough, but its move operations need clarifying. Consider the two possible actions that you need to take at each iteration of the `while` loop:

Move `theArray[firstUnknown]` into S_1. S_1 and the unknown region are, in general, not adjacent: S_2 is between the two regions. However, you can perform the required move efficiently. You swap `theArray[firstUnknown]` with the first item of S_2—which is `theArray[lastS1 + 1]`, as Figure 10-16 illustrates. Then you increment `lastS1` by 1. The item that was in `theArray[firstUnknown]` will then be at the rightmost position of S_1. What about the item of S_2 that was moved to `theArray[firstUnknown]`? If you increment `firstUnknown` by 1, that item becomes the rightmost member of S_2. Thus, you should perform the following steps to move `theArray[firstUnknown]` into S_1:

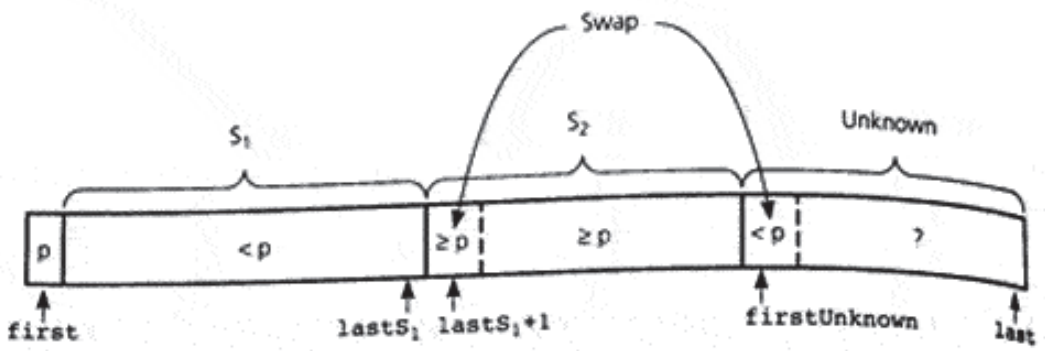


FIGURE 10-16

Moving `theArray[firstUnknown]` into S_1 by swapping it with `theArray[lastS1+1]` and by incrementing both `lastS1` and `firstUnknown`

```
Swap theArray[firstUnknown] with theArray[lastS1+1]
Increment lastS1
Increment firstUnknown
```

This strategy works even when S_2 is empty. In that case, `lastS1` + 1 equals `firstUnknown`, and thus the swap simply exchanges an item with itself. *This move preserves the invariant.*

Move `theArray[firstUnknown]` into S_2. This move is simple to accomplish. Recall that the rightmost boundary of the region S_2 is at position `firstUnknown` – 1; that is, regions S_2 and the unknown region are adjacent, as Figure 10-17 illustrates. Thus, to move `theArray[firstUnknown]` into S_2, simply increment `firstUnknown` by 1: S_2 expands to the right. *This move preserves the invariant.*

After you have moved all items from the unknown region into S_1 and S_2, one final task remains. You must place the pivot between the segments S_1 and S_2. Observe that `theArray[lastS1]` is the rightmost item in S_1. By interchanging this item with the pivot, which is `theArray[first]`, you will place the pivot in its correct location. Then the statement

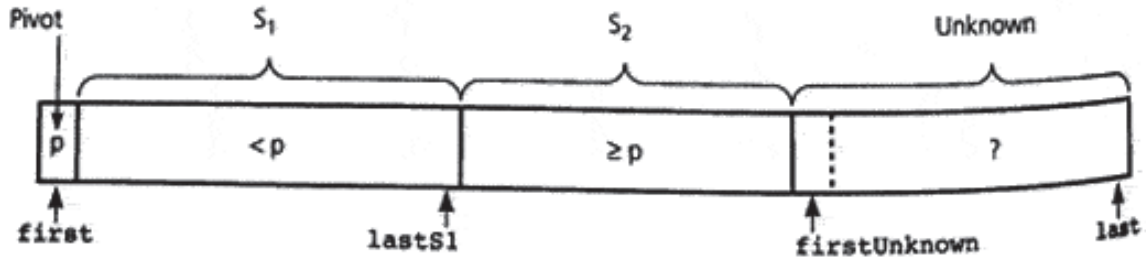


FIGURE 10-17

Moving `theArray[firstUnknown]` into S_2 by incrementing `firstUnknown`

```
return lastS1
```

returns the location of the pivot. You can use this index to determine the boundaries of S_1 and S_2. Figure 10-18 traces the partition algorithm for an array of six integers when the pivot is the first item.

Before continuing the implementation of quicksort, we will establish the correctness of the partition algorithm by using invariants. Again, the loop invariant for the algorithm is

> *All items in S_1 (`theArray[first+1..lastS1]`) are less than the pivot, and all items in S_2 (`theArray[lastS1+1..firstUnknown-1]`) are greater than or equal to the pivot.*

Recall that when you use invariants to establish the correctness of an iterative algorithm, a four-step process is required:

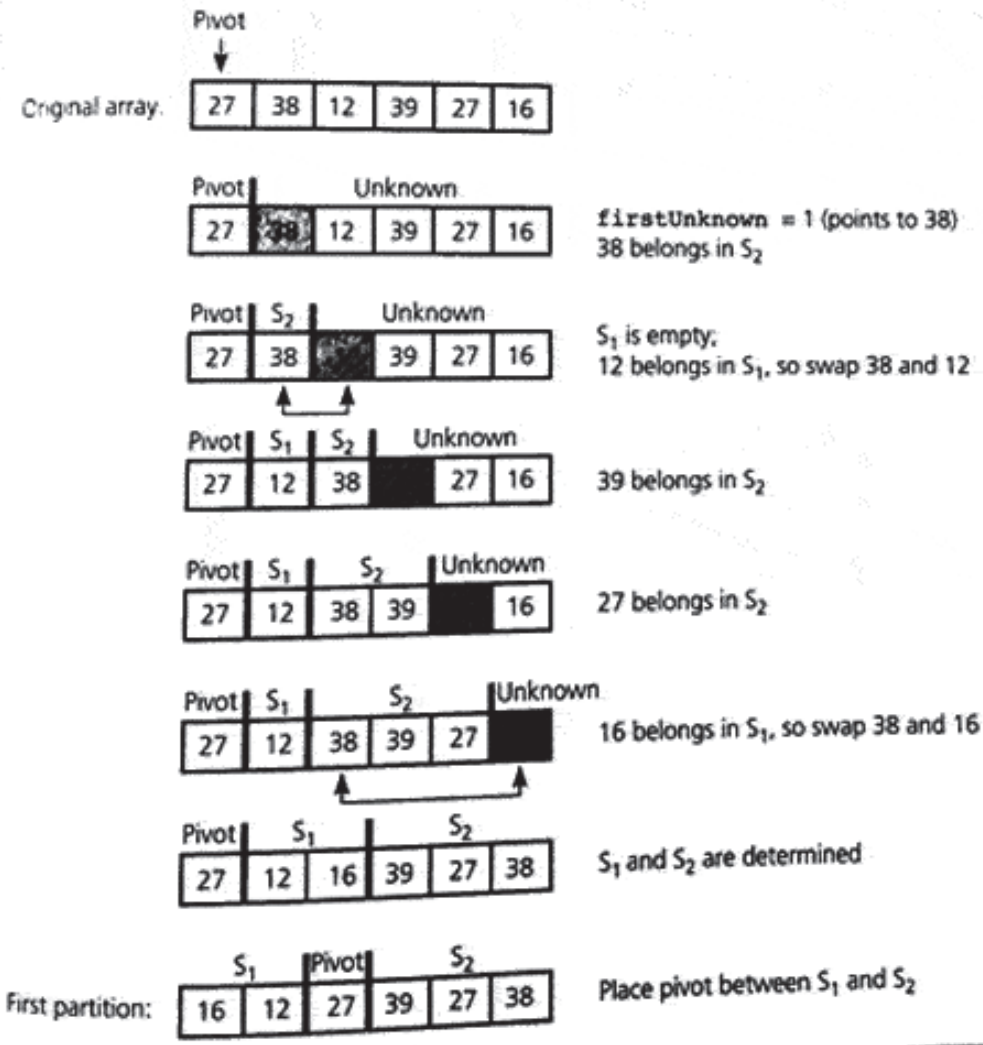


FIGURE 10-18

Developing the first partition of an array when the pivot is the first item

The proof that the partition algorithm is correct uses an invariant and requires four steps

1. The invariant must be true initially, before the loop begins execution. In the partition algorithm, before the loop that swaps array items is entered, the pivot is `theArray[first]`, the unknown region is `theArray[first+1..last]`, and S_1 and S_2 are empty. The invariant is clearly true initially.

2. An execution of the loop must preserve the invariant. That is, if the invariant is true before any given iteration of the loop, you must show that it is true after the iteration. In the partition algorithm, at each iteration of the loop a single item moves from the unknown region into either S_1 or S_2, depending on whether or not the item is less than the pivot. Thus, if the invariant was true before the move, it will remain true after the move.

3. The invariant must capture the correctness of the algorithm. That is, you must show that if the invariant is true when the loop terminates, the algorithm is correct. In the partition algorithm, the termination condition is that the unknown region is empty. But if the unknown region is empty, each item of `theArray[first+1..last]` must be in either S_1 or S_2—in which case the invariant implies that the partition algorithm has done what it was supposed to do.

4. The loop must terminate. That is, you must show that the loop will terminate after a finite number of iterations. In the partition algorithm, the size of the unknown region decreases by 1 at each iteration. Therefore, the unknown region becomes empty after a finite number of iterations, and thus the termination condition for the loop will be met.

The following Java methods implement the quicksort algorithm. The method `choosePivot` enables you to try various pivots easily. To sort an array `theArray` of `n` items, you invoke the method `quicksort` by writing `quicksort(theArray, 0, n-1)`.

```
private static <T extends Comparable<? super T>>
        void choosePivot(T[] theArray, int first, int last) {
// ------------------------------------------------------------
// Chooses a pivot for quicksort's partition algorithm and
// swaps it with the first item in an array.
// Precondition: theArray[first..last] where first <= last.
// Postcondition: theArray[first] is the pivot.
// ------------------------------------------------------------
// Implementation left as an exercise.
}  // end choosePivot
```

```
private static <T extends Comparable<? super T>>
        int partition(T[] theArray, int first, int last) {
    -----------------------------------------------------------
    Partitions an array for quicksort.
    Precondition: theArray[first..last] where first <= last.
    Postcondition: Returns the index of the pivot element of
    theArray[first..last]. Upon completion of the method,
    this will be the index value lastS1 such that
       S1 = theArray[first..lastS1-1] <  pivot
            theArray[lastS1]          == pivot
       S2 = theArray[lastS1+1..last]  >= pivot
    Calls: choosePivot.
    -----------------------------------------------------------
      tempItem is used to swap elements in the array
   T tempItem;
      place pivot in theArray[first]
   choosePivot(theArray, first, last);
   T pivot = theArray[first];   // reference pivot

      initially, everything but pivot is in unknown
   int lastS1 = first;          // index of last item in S1

      move one item at a time until unknown region is empty
      firstUnknown is the index of first item in unknown region

   for (int firstUnknown = first + 1; firstUnknown <= last;
                                      ++firstUnknown) {
     // Invariant: theArray[first+1..lastS1] < pivot
     //            theArray[lastS1+1..firstUnknown-1] >= pivot
     // move item from unknown to proper region
     if (theArray[firstUnknown].compareTo(pivot) < 0) {
       // item from unknown belongs in S1
       ++lastS1;
       tempItem = theArray[firstUnknown];
       theArray[firstUnknown] = theArray[lastS1];
       theArray[lastS1] = tempItem;
     }  // end if
   // else item from unknown belongs in S2
   }  // end for

   // place pivot in proper position and mark its location
   tempItem = theArray[first];
   theArray[first] = theArray[lastS1];
   theArray[lastS1] = tempItem;
   return lastS1;
}  // end partition
```

```
public static <T extends Comparable<? super T>>
       void quickSort(T[] theArray, int first, int last) {

  // ------------------------------------------------------
  // Sorts the items in an array into ascending order.
  // Precondition: theArray[first..last] is an array.
  // Postcondition: theArray[first..last] is sorted.
  // Calls: partition.
  // ------------------------------------------------------

  int pivotIndex;

  if (first < last) {
      create the partition: S1, Pivot, S2
    pivotIndex = partition(theArray, first, last);

      sort regions S1 and S2
    quickSort(theArray, first, pivotIndex-1);
    quickSort(theArray, pivotIndex+1, last);
  }  // end if
}  // end quickSort
```

In the analysis to follow, you will learn that it is desirable to avoid a pivot that makes either S_1 or S_2 empty. A good choice of pivot is one that is near the median of the array items. Exercise 20 at the end of this chapter considers this choice of pivot.

As you can see, *quicksort* and *mergesort* are similar in spirit, but whereas *quicksort* does its work before its recursive calls, *mergesort* does its work after its recursive calls. That is, while *quicksort* has the form

```
+quicksort(inout theArray:ItemArray,
           in first:integer, in last:integer)

 if (first < last) {
   Prepare theArray for recursive calls
   quicksort(S1 region of theArray)
   quicksort(S2 region of theArray)
      } // end if
```

mergesort has the general form

```
+mergesort(inout theArray:ItemArray,
           in first:integer, in last:integer)
```

```
if (first < last) {
  mergesort(Left half of theArray)
  mergesort(Right half of theArray)
  Tidy up array after the recursive calls
}  // end if
```

The preparation in `quicksort` is to partition the array into regions S_1 and S_2. The algorithm then sorts S_1 and S_2 independently, because every item in S_1 belongs to the left of every item in S_2. In `mergesort`, on the other hand, no work is done before the recursive calls: The algorithm sorts each half of the array with respect to itself. However, the algorithm must still deal with the interaction between the items in the two halves. That is, the algorithm must merge the two halves of the array after the recursive calls.

Analysis. The major effort in the `quicksort` method occurs during the partitioning step. As you consider each item in the unknown region, you compare `theArray[firstUnknown]` with the pivot and move `theArray[firstUnknown]` into either S_1 or S_2. It is possible for one of S_1 or S_2 to remain empty. For example, if the pivot is the smallest item in the array segment, S_1 will remain empty. This occurrence is the worst case because S_2 decreases in size by only 1 at each recursive call to `quicksort`. Thus, the maximum number of recursive calls to `quicksort` will occur.

Notice what happens when the array is already sorted into ascending order and you choose the first array item as the pivot. Figure 10-19 shows the results of the first call to `partition` for this situation. The pivot is the smallest item in the

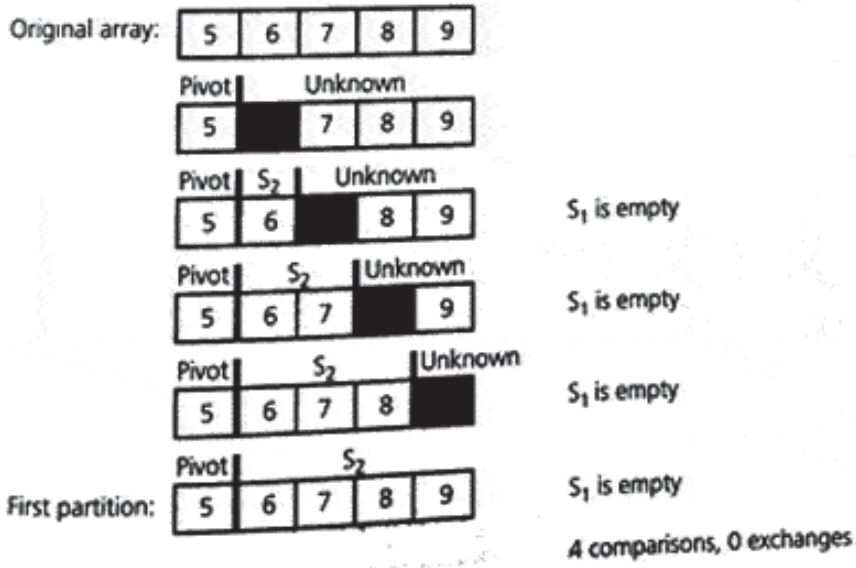


FIGURE 10-19

A worst-case partitioning with `quicksort`

quicksort is slow when the array is already sorted and you choose the smallest item as the pivot

array, and S_1 remains empty. In this case, `partition` requires $n - 1$ comparisons to partition the n items in this array. On the next recursive call to `quicksort`, `partition` is passed $n - 1$ items, so it will require $n - 2$ comparisons to partition them. Again, S_1, will remain empty. Because the array segment that `quicksort` considers at each level of recursion decreases in size by only 1, $n - 1$ levels of recursion are required. Therefore, `quicksort` requires

$$1 + 2 + \cdots + (n - 1) = n * (n - 1)/2$$

comparisons. However, recall that a move into S_2 does not require an exchange of array items; it requires only a change in the index `firstUnknown`.

Similarly, if S_2 remains empty at each recursive call, $n * (n - 1)/2$ comparisons are required. In addition, however, an exchange is necessary to move each unknown item to S_1. Thus, $n * (n - 1)/2$ exchanges are necessary. (Again, each exchange requires three data moves.) Thus, you can conclude that `quicksort` is $O(n^2)$ in the worst case.

In contrast, Figure 10-20 shows an example in which S_1 and S_2 contain the same number of items. In the average case, when S_1 and S_2 contain the same—or nearly the same—number of items arranged at random, fewer recursive calls to `quicksort` occur. As in the previous analysis of `mergesort`, you can conclude that there are either $\log_2 n$ or $1 + \log_2 n$ levels of recursive calls to `quicksort`. Each call to `quicksort` involves m comparisons and at most m exchanges, where m is the number of items in the subarray to be sorted. Clearly $m \leq n - 1$.

Quicksort. Worst case: $O(n^2)$, average case $O(n \cdot \log_2 n)$

A formal analysis of `quicksort`'s average-case behavior would show that it is $O(n * \log_2 n)$. Thus, on large arrays you can expect `quicksort` to run significantly faster than `insertionSort`, although in its worst case, `quicksort` will require roughly the same amount of time as `insertionSort`.

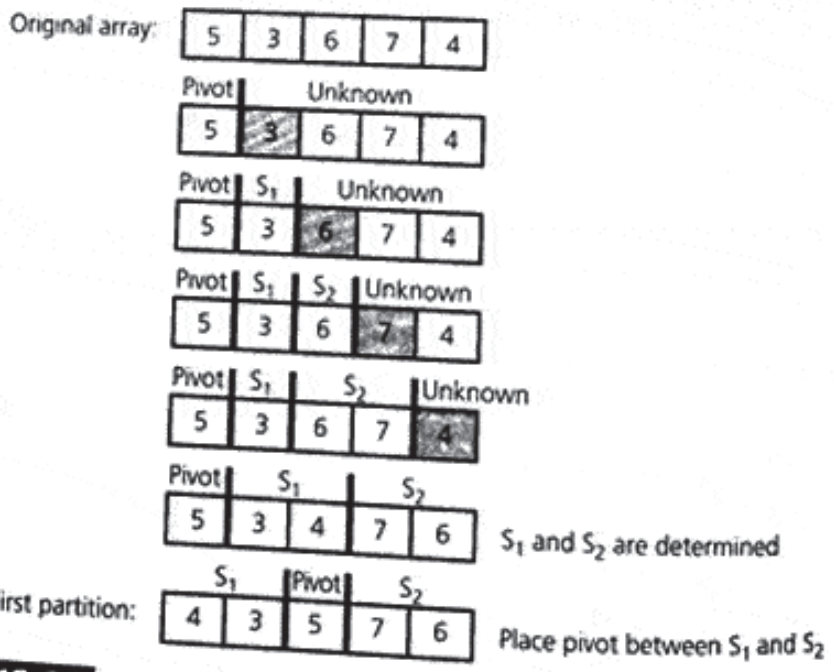


FIGURE 10-20

An average-case partitioning with `quicksort`

It might seem surprising, then, that `quicksort` is often used to sort large arrays. The reason for `quicksort`'s popularity is that it is usually extremely fast in practice, despite its unimpressive theoretical worst-case behavior. Although a worst case situation is not typical, even if the worst case occurs, `quicksort`'s performance is acceptable for moderately large arrays.

The fact that `quicksort`'s average-case behavior is far better than its worst-case behavior distinguishes it from the other sorting algorithms considered in this chapter. If the original arrangement of data in the array is "random," `quicksort` performs at least as well as any known sorting algorithm that involves comparisons. Unless the array is already ordered, `quicksort` is best.

The efficiency of `mergesort` is somewhere between the possibilities for `quicksort`: Sometimes `quicksort` is faster, and sometimes `mergesort` is faster. While the worst-case behavior of `mergesort` is of the same order of magnitude as `quicksort`'s average-case behavior, in most situations `quicksort` will run somewhat faster than `mergesort`. However, in its worst case, `quicksort` will be significantly slower than `mergesort`.

Radix Sort

The final sorting algorithm in this chapter is included here because it is quite different from the others.

Imagine one last time that you are sorting a hand of cards. This time you pick up the cards one at a time and arrange them by rank into 13 possible groups in this order: 2, 3, . . . , 10, J, Q, K, A. Combine these groups and place the cards face down on the table so that the 2s are on top and the aces are on the bottom. Now pick up the cards one at a time and arrange them by suit into four possible groups in this order: clubs, diamonds, hearts, and spades. When taken together, the groups result in a sorted hand of cards.

A **radix sort** uses this idea of forming groups and then combining them to sort a collection of data. The sort treats each data item as a character string. As a first simple example of a radix sort, consider this collection of three-letter strings:

ABC, XYZ, BWZ, AAC, RLT, JBX, RDT, KLT, AEO, TLJ

The sort begins by organizing the data according to their rightmost (least significant) letters. Although none of the strings ends in A or B, two strings end in C. Place those two strings into a group. Continuing through the alphabet, you form the following groups:

(ABC, AAC) (TLJ) (AEO) (RLT, RDT, KLT) (JBX) (XYZ, BWZ)

Group strings by rightmost letter

The strings in each group end with the same letter, and the groups are ordered by that letter. The strings within each group retain their relative order from the original list of strings.

Now combine the groups into one as follows. Take the items in the first group in their present order, follow them with the items in the second group in their present order, and so on. The following group results:

Combine groups

ABC, AAC, TLJ, AEO, RLT, RDT, KLT, JBX, XYZ, BWZ

Next, form new groups as you did before, but this time use the middle letter of each string instead of the last letter:

Group strings by middle letter

(AAC) (ABC, IBX) (RDT) (AEO) (TLI, RLT, KLT) (BWZ) (XYZ)

Now the strings in each group have the same middle letter, and the groups are ordered by that letter. As before, the strings within each group retain their relative order from the previous group of all strings.

Combine these groups into one group, again preserving the relative order of the items within each group:

Combine groups

AAC, ABC, IBX, RDT, AEO, TLI, RLT, KLT, BWZ, XYZ

Now form new groups according to the first letter of each string:

Group strings by first letter

(AAC, ABC, AEO) (BWZ) (IBX) (KLT) (RDT, RLT) (TLI) (XYZ)

Finally, combine the groups, again maintaining the relative order within each group:

Sorted strings

AAC, ABC, AEO, BWZ, IBX, KLT, RDT, RLT, TLI, XYZ

The strings are now in sorted order.

In the previous example, all character strings had the same length. If the character strings have varying lengths, you can treat them as if they were the same length by padding them on the right with blanks as necessary.

To sort numeric data, the radix sort treats a number as a character string. You can treat numbers as if they were padded on the left with zeros, making them all appear to be the same length. You then form groups according to the rightmost digits, combine the groups, form groups according to the next-to-last digits, combine them, and so on, just as you did in the previous example. Figure 10-21 shows a radix sort of eight integers.

0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150	Original integers
(1560, 2150) (1061) (0222) (0123, 0283) (2154, 0004)	Grouped by fourth digit
1560, 2150, 1061, 0222, 0123, 0283, 2154, 0004	Combined
(0004) (0222, 0123) (2150, 2154) (1560, 1061) (0283)	Grouped by third digit
0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283	Combined
(0004, 1061) (0123, 2150, 2154) (0222, 0283) (1560)	Grouped by second digit
0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560	Combined
(0004, 0123, 0222, 0283) (1061, 1560) (2150, 2154)	Grouped by first digit
0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154	Combined (sorted)

FIGURE 10-21

A radix sort of eight integers

The following pseudocode describes the algorithm for a radix sort of n decimal integers of d digits each:

```
radixSort(inout theArray:ItemArray,
          in n:integer, in d:integer)
  Sorts n d-digit integers in the array theArray.

 for (j = d down to 1) {
   Initialize 10 groups to empty
   Initialize a counter for each group to 0
   for (i = 0 through n-1) {
     k = j^th digit of theArray[i]
     Place theArray[i] at the end of group k
     Increase k^th counter by 1
   }  // end for i

   Replace the items in theArray with all the
       items in group 0, followed by all the items
       in group 1, and so on.
 }  // end for j
```

Analysis. From the pseudocode for the radix sort, you can see that this algorithm requires n moves each time it forms groups and n moves to combine them again into one group. The algorithm performs these $2 * n$ moves d times. Therefore, the radix sort requires $2 * n * d$ moves to sort n strings of d characters each. However, notice that no comparisons are necessary. Thus, radix sort is $O(n)$.

A Comparison of Sorting Algorithms

Figure 10-22 summarizes the worst-case and average-case orders of magnitude for the sorting algorithms that appear in this chapter. For reference purposes, two other algorithms—treesort and heapsort—are

	Worst case	Average case
Selection sort	n^2	n^2
Bubble sort	n^2	n^2
Insertion sort	n^2	n^2
Mergesort	n * log n	n * log n
Quicksort	n^2	n * log n
Radix sort	n	n
Treesort	n^2	n * log n
Heapsort	n * log n	n * log n

FIGURE 10-22

Approximate growth rates of time required for eight sorting algorithms

included here, even though you will not study them until Chapters 11 and 12, respectively.

The Java Collections Framework Sort Algorithm

The Java Collections Framework provides numerous polymorphic algorithms. These algorithms appear in the class `java.util.Collections`. Note that this is a different class than the `Collection` class used as the root interface in the collection hierarchy. The `Collections` class provides only static methods that operate on or return collections.

Typically, these static methods take an argument that is the collection on which the method is to be performed. Many of the methods have a parameter that is of type `Collection` or `List`.

The sort algorithm used in the JCF is a slightly optimized version of the *mergesort* algorithm. As we have seen in our analysis of *mergesort*, it is guaranteed to run in $n*log(n)$ time. Also note that *mergesort* is stable; the elements with the same value are guaranteed to remain in the same relative order. This can be important if you are sorting the same list repeatedly on different attributes. For example, if a threaded discussion is sorted by date and then sorted by sender, the user expects the now-contiguous list of messages from a given sender will still be sorted by date. This will only happen if the algorithm used for the second sort is stable.

There are actually two sort methods provided in the `Collections` class:

```
static <T extends Comparable<? super T>> void sort(List<T> list)
  // Sorts the specified list into ascending order, according
  // to the natural ordering of its elements.

static <T> void sort(List<T> list, Comparator<? super T> c)
  // Sorts the specified list according to the order induced
  // by the specified comparator.
```

The first form takes a `List<T>`, where `T` or one of its super classes should implement the `Comparable` interface. It sorts the items into ascending order based upon the natural ordering of the elements. Here is a simple program that demonstrates how to use this sort to alphabetically order a list of names:

```
public static void main(String args[]) {
  String[] names = {"Janet", "Michael", "Andrew", "Kate",
                    "Sarah", "Regina", "Rachael", "Allie"};
  List<String> l = Arrays.asList(names);
  Collections.sort(l);
  System.out.println(l);
} // end main
```

The method `Arrays.asList` takes an array argument and returns it as a serializable `List` that is subsequently sent to the `Collections.sort` method. This program produces the following output:

```
[Allie, Andrew, Janet, Kate, Michael, Rachael, Regina, Sarah]
```

The second form of sort takes two parameters, the first is a `List<T>`, this time with no restrictions on `T`, and a second parameter that is a `Comparator` object. Note the `Comparator` interface is defined as follows:

```
public interface Comparator<T> {

  int compare(T o1, T o2);
    // Compares its two arguments for order.

  boolean equals(Object obj);
    // Indicates whether some other object is "equal to" this
    // Comparator.
}  // end Comparator
```

As mentioned earlier in the chapter, use of the `Comparator` object imposes a total ordering on some collection of objects. This may be the same as the natural ordering that is usually implemented using the `Comparable` interface. To utilize the `sort` method with a `Comparator` object requires an implementation of the `Comparator` interface for the data type of the objects in the collection.

For example, suppose we have a simple collection of `Person` objects, where each object contains the name and age for a person as follows:

```
class Person implements Serializable {
  private String name;
  private int age;

  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  } // end constructor
```

```
  public String getName() {
    return name;
  } // end getName

  public int getAge() {
    return age;
  } // end getAge

  public String toString() {
   return name + " - " + age;
  } // end toString
} //end Person
```

The *Person* class implements the *Serializable* interface since we plan to use the *Arrays.asList* method again to create a list. We will demonstrate two different comparators for the *Person* class, one to compare by name, the other by age. The *Comparator* class is a generic class, so when we implement it, the data type *Person* will be provided as the generic parameter. Each comparator requires the definition of a *compare* method and an *equals* method. The *compare* method is implemented to impose and ordering on two *Person* objects based upon some criteria, name or age. The *equals* method is intended to allow you to determine if you have two comparators that are the same. The implementation shown here simply checks to see if the comparator objects are the same object, not the same type of object. Other interpretations are possible.

Here is the definition of the *NameComparator* class; the compare method is based solely on the *name* field of the *Person* object:

```
import java.util.Comparator;
import java.io.Serializable;

class NameComparator implements Comparator<Person>, Serializable {

  public int compare(Person o1, Person o2) {
    // Compares its two arguments for order by name.
    return o1.getName().compareTo(o2.getName());
  } // end compare

  public boolean equals(Object obj) {
    // Simply checks to see if we have the same object
    return this==obj;
  } // end equals

} // end NameComparator
```

The definition of the `AgeComparator` class is quite similar, but the compare method is based solely on the `age` field of the `Person` object:

```
import java.util.Comparator;
import java.io.Serializable;

class AgeComparator implements Comparator<Person>, Serializable {
  public int compare(Person o1, Person o2) {
    // Returns the difference:
    // if positive, age of o1 person is greater than o2 person
    // if zero, the ages are equal
    // if negative, age of o1 person is less than o2 person
    return o1.getAge() - o2.getAge();
  } // end compare

  public boolean equals(Object obj) {
    // Simply checks to see if we have the same object
    return this==obj;
  } // end equals
} // end AgeComparator
```

Note that the comparators implement the *Serializable* interface. It is considered good programming practice to do this as it is quite possible that these comparator objects may be part of other serializable data structures.

Now that these comparator classes exist, comparator objects are instantiated for use in the sort method. For example:

```
public static void main(String args[]) {
  NameComparator nameComp = new NameComparator();
  AgeComparator ageComp = new AgeComparator();

  Person[] p = new Person[5];
  p[0] = new Person("Michael", 45);
  p[1] = new Person("Janet", 39);
  p[2] = new Person("Sarah", 17);
  p[3] = new Person("Kate", 20);
  p[4] = new Person("Andrew", 20);
  List<Person> list = Arrays.asList(p);

  System.out.println("Sorting by age:");
  Collections.sort(list, ageComp);
  System.out.println(list);

  System.out.println("Sorting by name:");
  Collections.sort(list, nameComp);
  System.out.println(list);
```

```
    System.out.println("Now sorting by age, after sorting by name:");
    Collections.sort(list, ageComp);
    System.out.println(list);
  } // end main
```

This program produces the following output:

```
Sorting by age:
[Sarah - 17, Kate - 20, Andrew - 20, Janet - 39, Michael - 45]
Sorting by name:
[Andrew - 20, Janet - 39, Kate - 20, Michael - 45, Sarah - 17]
Now sorting by age, after sorting by name:
[Sarah - 17, Andrew - 20, Kate - 20, Janet - 39, Michael - 45]
```

Note that the sort is stable—it maintains the order of equal elements. This is evident by observing that the first sort by age kept the names with the same age in their original order. After sorting by name first, then by age, the names are now displayed by age with the names in alphabetical order.

Summary

1. Order-of-magnitude analysis and Big O notation measure an algorithm's time requirement as a function of the problem size by using a growth-rate function. This approach enables you to analyze the efficiency of an algorithm without regard for such factors as computer speed and programming skill that are beyond your control.
2. When you compare the inherent efficiency of algorithms, you examine their growth-rate functions when the problems are large. Only significant differences in growth-rate functions are meaningful.
3. Worst-case analysis considers the maximum amount of work an algorithm will require on a problem of a given size, while average-case analysis considers the expected amount of work that it will require.
4. You can use order-of-magnitude analysis to help you choose an implementation for an abstract data type. If your application frequently uses particular ADT operations, your implementation should be efficient for at least those operations.
5. Selection sort, bubble sort, and insertion sort are all $O(n^2)$ algorithms. Although, in a particular case, one might be faster than another, for large problems they all are slow.
6. Quicksort and mergesort are two very efficient recursive sorting algorithms. In the "average" case, quicksort is among the fastest known sorting algorithms. However, quicksort's worst-case behavior is significantly slower than mergesort's. Fortunately, quicksort's worst case rarely occurs in practice. Actual execution time for mergesort is not quite as fast as quicksort in the average case, even though they have the same order. However, mergesort's performance is consistently good in all cases. Mergesort has the disadvantage of requiring extra storage equal to the size of the array to be sorted.

Questions

- In general, you should avoid analyzing an algorithm solely by studying the running times of a specific implementation. Running times are influenced by such factors as programming style, the particular computer, and the data on which the program is run.
- When comparing the efficiency of various solutions, look only at significant differences. This rule is consistent with the multidimensional view of the cost of a computer program.
- While manipulating the Big O notation, remember that $O(f(n))$ represents an inequality. It is not a function but simply a notation that means "is of order $f(n)$" or "is order $f(n)$."
- If a problem is small, do not overanalyze it. In such a situation, the primary concern should be simplicity. For example, if you are sorting an array that contains only a small number of items—say, fewer than 25—a simple $O(n^2)$ algorithm such as an insertion sort is appropriate.
- If you are sorting a very large array, an $O(n^2)$ algorithm is probably too inefficient to use.
- Quicksort is appropriate when you are confident that the data in the array to be sorted is arranged randomly. Although quicksort's worst-case behavior is $O(n^2)$, the worst case rarely occurs in practice.

Self-Test Exercises

1. How many comparisons of array items do the following loops contain?

```
int temp;
for (j = 1; j <= n-1; ++j) {
  i = j + 1;
  do {
    if (theArray[i] < theArray[j]) {
      temp = theArray[i];
      theArray[i] = theArray[j];
      theArray[j] = temp;
    }  // end if
    ++i;
  }  while (i <= n);
}  // end for
```

2. Repeat Self-Test Exercise 1, replacing the statement `i = j + 1` with `i = j`.
3. What order is an algorithm that has as a growth-rate function
   a. $8 * n^3 - 9 * n^2$ b. $7 * \log_2 n + 20$ c. $7 * \log_2 n * n$
4. Consider a sequential search of n data items.
   a. If the data items are sorted into descending order, how can you determine that your desired item is not in the data collection without always making n comparisons?

b. What is the order of the sequential search algorithm when the desired item is not in the data collection? Do this for both sorted and unsorted data, and consider the best, average, and worst cases.

c. Show that if the sequential search algorithm finds the desired item in the data collection, the algorithm's order does not depend upon whether or not the data items are sorted.

5. Trace the selection sort as it sorts the following array into ascending order: 80 40 25 20 30 60.
6. Repeat Self-Test Exercise 5, but instead sort the array into descending order.
7. Trace the bubble sort as it sorts the following array into ascending order: 80 40 25 20 30 60.
8. Trace the insertion sort as it sorts the array in Self-Test Exercise 7 into ascending order.
9. Show that the mergesort algorithm satisfies the four criteria of recursion that Chapter 3 describes.
10. Trace quicksort's partitioning algorithm for an ascending sort as it partitions the following array. Use the first item as the pivot.

    39 12 16 38 40 27

11. Suppose that you sort a large array of integers by using mergesort. Next you use a binary search to determine whether a given integer occurs in the array. Finally, you display all the integers in the sorted array.

    a. Which algorithm is faster, in general: the mergesort or the binary search? Explain in terms of Big O notation.

    b. Which algorithm is faster, in general: the binary search or displaying the integers? Explain in terms of Big O notation.

Exercises

1. What is the order of each of the following tasks in the worst case?

    a. Computing the sum of the first half of an array of n items

    b. Initializing each element of an array `items` to 1

    c. Displaying every other integer in a linked list of n nodes

    d. Displaying all n names in a circular linked list

    e. Displaying the third element in a linked list

    f. Displaying the last integer in a linked list of n nodes

    g. Searching an array of n integers for a particular value by using a binary search

    h. Sorting an array of n integers into ascending order by using a mergesort

2. Why do we include the variable `sorted` in the implementation of the bubble

3. For queues and stacks presented earlier in this text, three implementations were provided—a reference based implementation, an array based implementation, and a list based implementation. For each of these implementations, what is the order of each of the following tasks in the worst case?

   a. Adding an item to a stack of n items

   b. Adding an item to a queue of n items

4. Find an array that makes the bubble sort exhibit its worst behavior.

5. Suppose that your implementation of a particular algorithm appears in Java as

```
for (int pass = 1; pass <= n; ++pass) {
  for (int index = 0; index < n; ++index) {
    for (int count = 1; count < 10; ++count) {
      . . .
    } // end for
  } // end for
} // end for
```

   The previous code shows only the repetition in the algorithm, not the computations that occur within the loops. These computations, however, are independent of n. What is the order of the algorithm? Justify your answer.

6. Consider the following Java method `f`. Do not be concerned with `f`'s purpose.

```
public static void f(int[] theArray, int n) {
 int temp;
  for (int j = 0; j < n; ++j) {
    int i = 0;
    while (i <= j) {
      if (theArray[i] < (theArray[j])) {
        temp = theArray[i];
        theArray[i] = theArray[j];
        theArray[j] = temp;
      }  // end if
      ++i;
    }  // end while
  }  // end for
}  // end f
```

   How many comparisons does `f` perform?

7. For large arrays and in the worst case, is selection sort faster than insertion sort? Explain.

8. Show that any polynomial $f(x) = c_n x^n + c_{n-1}x^{n-1} + \cdots + c_1 x + c_0$ is $O(x^n)$.

9. Show that for all constants a, $b > 1$, $f(n)$ is $O(\log_a n)$ if and only if it is $O(\log_b n)$. Thus, you can omit the base when you write $O(\log n)$. Hint: Use the identity $\log_a n = \log_b n / \log_b a$ for all constants a, $b > 1$.

10. This chapter's analysis of selection sort ignored the cost of the statements that manipulate array indexes. Revise this analysis to account for these statements. Show that the algorithm is still $O(n^2)$.

11. Trace the insertion sort as it sorts the following array into ascending order:
80 40 25 20 30 60

12. Trace the selection sort as it sorts the following array into ascending order:
8 11 23 1 20 33

13. Trace the bubble sort as it sorts the following array into descending order:
10 12 23 34 5

14. Apply the selection sort, bubble sort, and insertion sort to

    a. An inverted array: 9 7 5 3 1

    b. An ordered array: 1 3 5 7 9

15. How many comparisons would be needed to sort an array containing 100 elements using the bubble sort in

    a. the worst case?

    b. the best case?

16. Write recursive versions of `selectionSort`, `bubbleSort`, and `insertionSort`.

17. One way computer speeds are measured is by the number of instructions they can perform per second. Assume that a comparison or a data move are each a single instruction. If the bubble sort is being used to sort 1,000,000 items in a worst case scenario, what is the approximate amount of time it takes to execute this sort on each of the following computers? Express your answer in days, hours, minutes, and seconds.

    a. An early computer that could only execute one thousand instructions per second

    b. A more recent computer that can execute one billion instructions per second

18. Trace the mergesort algorithm as it sorts the following array into ascending order. List the calls to `mergesort` and to `merge` in the order in which they occur.
80 40 25 20 30 60

19. When sorting an array by using mergesort,

    a. Do the recursive calls to `mergesort` depend on the values in the array, the number of items in the array, or both? Explain.

    b. In what step of `mergesort` are the items in the array actually swapped (that is, sorted)? Explain.

20. Trace the quicksort algorithm as it sorts the following array into ascending order. List the calls to `quicksort` and to `partition` in the order in which they occur.
80 40 25 20 30 60 15

21. Suppose that you remove the call to `merge` from the `mergesort` algorithm to obtain

```
+mystery(inout theArray:ItemArray,
         in first:integer, in last:integer)
// mystery algorithm for theArray[first..last].
```

```
if (first < last) {
  mid = (first + last) / 2
  mystery(theArray, first, mid)
  mystery(theArray, mid+1, last)
}  // end if
```

What does this new algorithm do?

22 You can choose any array item as the pivot for `quicksort`. You then interchange items so that your pivot is in `theArray[first]`.

a. One way to choose a pivot is to take the middle value of the three values `theArray[first]`, `theArray[last]`, and `theArray [(first + last)/2]`. How many comparisons are necessary to sort an array of size n if you always choose the pivot in this way?

b. If the actual median value could be chosen as the pivot at each step, how many comparisons are necessary to sort an array of size n?

23 The partition algorithm that `quicksort` uses moves one item at a time from the unknown region into the appropriate region S_1 or S_2. If the item to be moved belongs in region S_1, and if S_2 is empty, the algorithm will swap an array item with itself. Modify the partition algorithm to eliminate this unnecessary swapping. Does this change the order of the algorithm?

24 Use invariants to show that the method `selectionSort` is correct.

25 Describe an iterative version of `mergesort`. Define an appropriate invariant and show the correctness of your algorithm.

26 One criterion used to evaluate sorting algorithms is stability. A sorting algorithm is stable if it does not exchange items that have the same sort key. Thus, items with the same sort key (possibly differing in other ways) will maintain their positions relative to one another. For example, you might want to take an array of students sorted by name and sort it by year of graduation. Using a stable sorting algorithm to sort the array by year will ensure that within each year the students will remain sorted by name. Some applications mandate a stable sorting algorithm. Others do not. Which of the sorting algorithms described in this chapter are stable?

27 When we discussed the radix sort, we sorted a hand of cards by first ordering the cards by rank and then by suit. To implement a radix sort for this example, you could use two characters to represent a card, if you used T to represent a 10. For example, S2 is the 2 of spades and HT is the 10 of hearts.

a. Show a trace of the radix sort for the following cards:

S2, HT, D6, S4, C9, CJ, DQ, ST, HQ, DK

b. Suppose that you did not use T to represent a 10—that is, suppose that H10 is the 10 of hearts—and that you padded the two-character strings on the right with a blank to form three-character strings. How would a radix sort order the entire deck of cards in this case?

Programming Problems

1. Add counters to the methods `insertionSort` and `mergesort` to count the number of comparisons that are made and the number of exchanges that are performed. Run the two methods with arrays of various sizes. At what size does the difference in the number of comparisons become significant? The number of exhanges? How does this size compare with the size that the orders of these algorithms predict?

2. The method `quicksort` uses the method `choosePivot` to choose a pivot and place it into the first array location. Implement `choosePivot` in two ways:

   a. Always choose the first item in the array as pivot.

   b. Choose a pivot as Exercise 22a describes.

   Add a counter to the method `partition` that counts the number of comparisons that are made. Run `quicksort` with each pivot selection strategy and with arrays of various sizes. On what size array does the difference in the number of comparisons become significant? For which pivot selection strategy does the difference in the number of comparisons become significant?

3. a. Modify the partition algorithm for `quicksort` so that S_1 and S_2 will never be empty.

   b. Another partitioning strategy for `quicksort` is possible. Let an index `low` traverse the array segment `theArray [first..last]` from `first` to `last` and stop when it encounters the first item that is greater than the pivot item. Similarly, let a second index `high` traverse the array segment from `last` to `first` and stop when it encounters the first item that is smaller than the pivot item. Then swap these two items, increment `low`, decrement `high`, and continue until `high` and `low` meet somewhere in the middle. Implement this version of `quicksort` in Java. How can you ensure that the regions S_1 and S_2 are not empty?

   c. There are several variations of this partitioning strategy. What other strategies can you think of? How do they compare to the two that have been given?

4. Implement the radix sort of an array by using an ADT queue for each group.

5. Implement a radix sort of a linked list of integers.

6. a. Implement a mergesort of a linked list of integers.

   b. Implement any other sorting algorithms that are appropriate for a linked list.

7. You can sort a large array of integers that are in the range 1 to 100 by using an array `count` of 100 items to count the number of occurrences of each integer in the array. Fill in the details of this sorting algorithm, which is called a **bucket sort**, and write a Java method that implements it. What is the order of the bucket sort? Why is the bucket sort not useful as a general sorting algorithm?

8. Shellsort (named for its inventor, Donald Shell) is an improved insertion sort. Rather than always exchanging adjacent items—as in insertion sort—Shellsort can exchange items that are far apart in the array. Shellsort arranges the array so that every h^{th} item forms a sorted subarray. For every h in a decreasing sequence of

... the array. For example, if h is 5, every fifth item forms a ... Ultimately, if h is 1, the entire array will be sorted.
... sequence of h's begins at $n/2$ and halves n until it becomes 1. By ... and by replacing 1 with h and 0 with h-1 in `insertionSort`, ... method for Shellsort:

```
public static <T extends Comparable<? super T>>
      void shellSort(T[] theArray, int n) {
  int h;
  T nextItem;
  for (int h = n/2; h > 0; h = h/2) {
    for (int unsorted = h; unsorted < n; ++unsorted) {
      nextItem = theArray[unsorted];
      int loc = unsorted;
      while ((loc >= h) &&
             (theArray[loc-h].compareTo(nextItem) > 0)) {
        theArray[loc] = theArray[loc-h];
        loc = loc - h;
      } // end while
      theArray[loc] = nextItem;
    } // end for unsorted
  } // end for h
} // end shellSort
```

Add a counter to the methods `insertionSort` and `shellsort` that counts the number of comparisons that are made. Run the two methods with arrays of ... size. On what size does the difference in the number of comparisons become significant?

CHAPTER 11

Trees

The data organizations presented in previous chapters are linear in that items are one after another. The ADTs in this chapter organize data in a nonlinear, hierarchical form, whereby an item can have more than one immediate successor. In particular, this chapter discusses the specifications, implementations, and relative efficiency of the ADT binary tree and the ADT binary search tree. These ADTs are basic to the next three chapters.

General categories of data-management operations

The previous chapters discussed ADTs whose operations fit into at least one of these general categories:

- Operations that insert data into a data collection
- Operations that delete data from a data collection
- Operations that ask questions about the data in a data collection

The form of operations on position-oriented ADTs

The ADTs list, stack, and queue are all **position oriented,** and their operations have the form

- Insert a data item into the i^{th} *position* of a data collection.
- Delete a data item from the i^{th} *position* of a data collection.
- Ask a question about the data item in the i^{th} *position* of a data collection.

As you have seen, the ADT list places no restriction on the value of i, while the ADTs stack and queue do impose some restrictions. For example, the operations of the ADT stack are restricted to inserting into, deleting from, and asking a question about one end—the top—of the stack. Thus, although they differ with respect to the flexibility of their operations, lists, stacks, and queues manage an association between data items and *positions.*

The form of operations on value-oriented ADTs

The ADT sorted list is **value oriented.** Its operations are of the form

- Insert a data item containing the *value x.*
- Delete a data item containing the *value x.*
- Ask a question about a data item containing the *value x.*

Although these operations, like position-oriented operations, fit into the three general categories of operations listed earlier—they insert data, delete data, and ask questions about data—they are based upon *values* of data items instead of *positions.*

This chapter discusses two major ADTs: the binary tree and the binary search tree. As you will see, the binary tree is a position-oriented ADT, but it is not linear as are lists, stacks, and queues. Thus, you will not reference items in a binary tree by using a position number. Our discussion of the ADT binary tree provides an important background for the more useful binary search tree, which is a value-oriented ADT. Although a binary search tree is also not linear, it has operations similar to those of a sorted list, which is linear.

In the next chapter, you will see two more value-oriented ADTs: the ADT table and the ADT priority queue. The implementations of both of these ADTs can use the ideas presented in this chapter.

11.1 Terminology

You use trees to represent relationships. Previous chapters informally used tree diagrams to represent the relationships between the calls of a recursive algorithm. For example, the diagram of the *rabbit* algorithm's recursive calls in

Figure 3-11 of Chapter 3 is actually a tree. Each call to *rabbit* is represented by a box, or **node,** or **vertex,** in the tree. The lines between the nodes (boxes) are called **edges.** For this tree, the edges indicate recursive calls. For example, the edges from *rabbit*(7) to *rabbit*(6) and *rabbit*(5) indicate that subproblem *rabbit*(7) makes calls to *rabbit*(6) and *rabbit*(5).

Trees are hierarchical

All trees are **hierarchical** in nature. Intuitively, hierarchical means that a "parent-child" relationship exists between the nodes in the tree. If an edge is between node n and node m, and node n is above node m in the tree, then n is the **parent** of m, and m is a **child** of n. In the tree in Figure 11-1, nodes B and C are children of node A. Children of the same parent—for example, B and C—are called **siblings.** Each node in a tree has at most one parent, and exactly one node—called the **root** of the tree—has no parent. Node A is the root of the tree in Figure 11-1. A node that has no children is called a **leaf** of the tree. The leaves of the tree in Figure 11-1 are C, D, E, and F.

A subtree is any node and its descendants

The parent-child relationship between the nodes is generalized to the relationships **ancestor** and **descendant.** In Figure 11-1, A is an ancestor of D, and thus D is a descendant of A. Not all nodes are related by the ancestor or descendant relationship: B and C, for instance, are not so related. However, the root of any tree is an ancestor of every node in that tree. A **subtree** in a tree is any node in the tree together with all of its descendants. A **subtree of a node** n is a subtree rooted at a child of n. For example, Figure 11-2 shows a subtree of the tree in Figure 11-1. This subtree has B as its root and is a subtree of the node A.

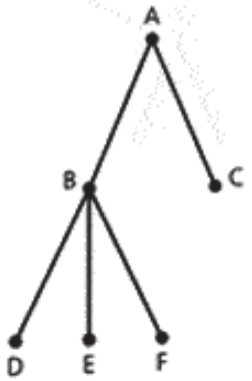


FIGURE 11-1
A general tree

FIGURE 11-2
A subtree of the tree in Figure 11-1

Because trees are hierarchical in nature, you can use them to represent information that itself is hierarchical in nature—for example, organization charts and family trees, as Figure 11-3 depicts. It may be disconcerting to discover, however, that the nodes in the family tree in Figure 11-3b that represent Caroline's parents (John and Jacqueline) are the children of the node that represents Caroline! That is, the nodes that represent Caroline's ancestors are the descendants of Caroline's node. It's no wonder that computer scientists often seem to be confused by reality.

Formally, a **general tree** T is a set of one or more nodes such that T is partitioned into disjoint subsets:

- A single node r, the root
- Sets that are general trees, called subtrees of r

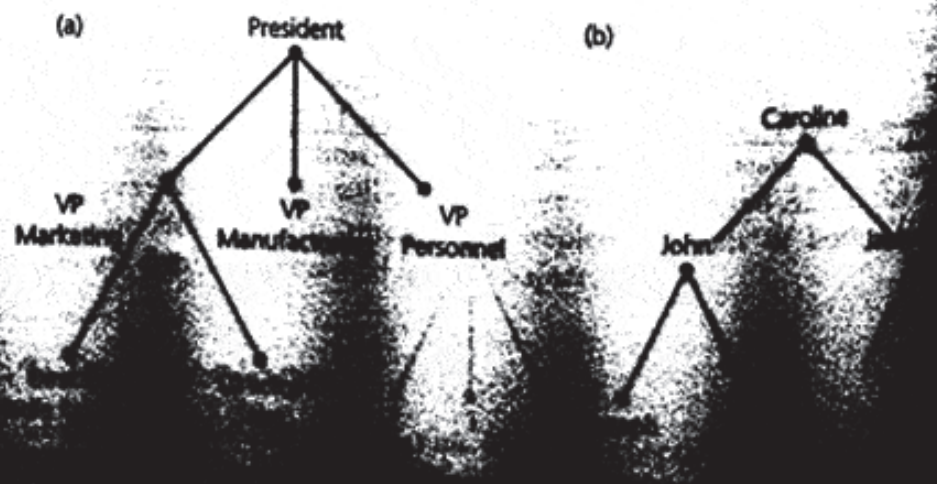
Thus, the trees in Figures 11-1 and 11-3a are general trees.

Formal definition of a binary tree

The primary focus of this chapter will be on binary trees. Formally, a **binary tree** is a set T of nodes such that either

- T is empty, or
- T is partitioned into three disjoint subsets:
  - A single node r, the root
  - Two possibly empty sets that are binary trees, called **left** and **right subtrees** of r

The trees in Figures 3-11a and 11-3b are binary trees. Notice that each node in a binary tree has no more than two children. A binary tree is not a special kind of general tree, because a binary tree can be empty, whereas a general tree cannot.

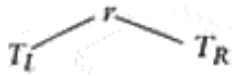


The following intuitive restatement of the definition of a binary tree is useful:

Intuitive definition of a binary tree

T is a binary tree if either

- T has no nodes, or
- T is of the form



where r is a node and T_L and T_R are both binary trees

Notice that the formal definition agrees with this intuitive one: If r is the root of T, then the binary tree T_L is the left subtree of node r and T_R is the right subtree of node r. If T_L is not empty, its root is the **left child** of r, and if T_R is not empty, its root is the **right child** of r. Notice that if both subtrees of a node are empty, that node is a leaf.

As an example of how you can use a binary tree to represent data in a hierarchical form, consider Figure 11-4. The binary trees in this figure represent algebraic expressions that involve the binary operators +, −, *, and /. To represent an expression such as $a - b$, you place the operator in the root node and the operands a and b into the left and right children, respectively, of the root. (See Figure 11-4a.) Figure 11-4b represents the expression $a - b/c$; a subtree represents the subexpression b/c. A similar situation exists in Figure 11-4c, which represents $(a - b) * c$. The leaves of these trees contain the expressions' operands, while other tree nodes contain the operators. Parentheses do not appear in these trees. The binary tree provides a hierarchy for the operations—that is, the tree specifies an unambiguous order for evaluating an expression.

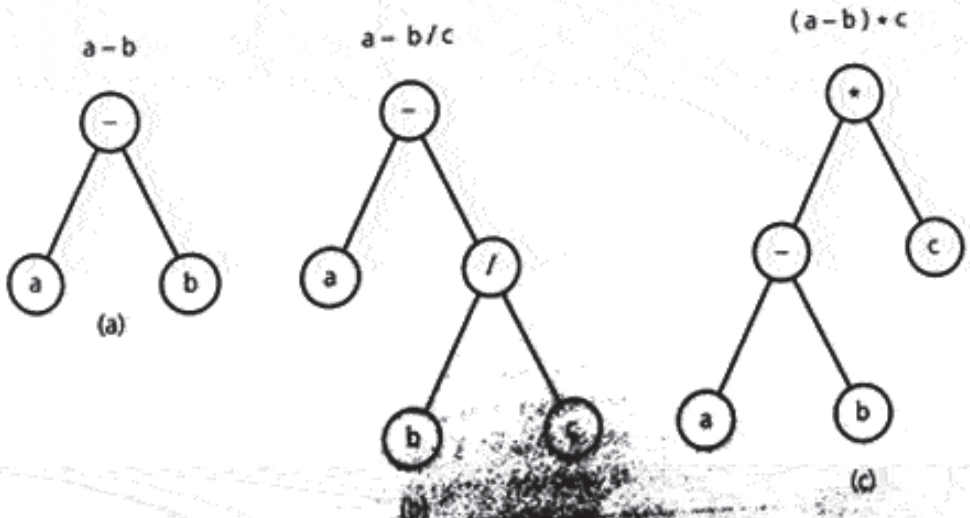


FIGURE 11-4
Binary trees that represent al

Properties of a binary search tree

The nodes of a tree typically contain values. A **binary search tree** is a binary tree that is in a sense sorted according to the values in its nodes. For each node n, a binary search tree satisfies the following three properties:

- n's value is greater than all values in its left subtree T_L.
- n's value is less than all values in its right subtree T_R.
- Both T_L and T_R are binary search trees.

Figure 11-5 is an example of a binary search tree. As its name suggests, a binary search tree organizes data in a way that facilitates searching it for a particular data item. Later, this chapter discusses binary search trees in detail.

The height of trees. Trees come in many shapes. For example, although the binary trees in Figure 11-6 all contain the same nodes, their structures are

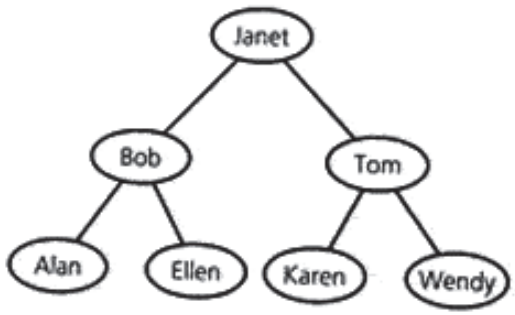


FIGURE 11-5
A binary search tree of names

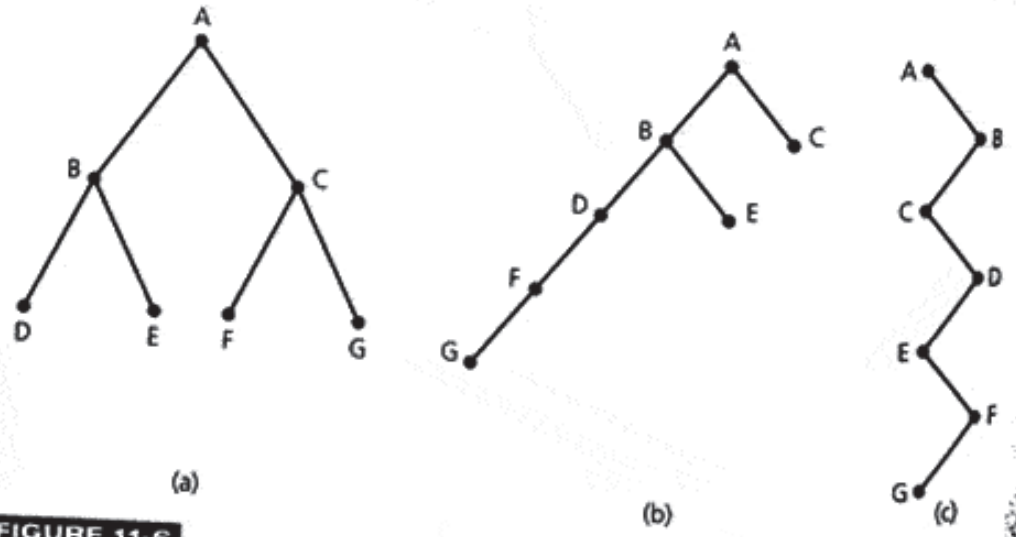


FIGURE 11-6
Binary trees with the same nodes but different heights

quite different. Although each of these trees has seven nodes, some are "taller" than others. The **height of a tree** is the number of nodes on the longest path from the root to a leaf. For example, the trees in Figure 11-6 have respective heights of 3, 5, and 7. Many people's intuitive notion of height would lead them to say that these trees have heights of 2, 4, and 6. Indeed, many authors define height to agree with this intuition. However, the definition of height used in this book leads to a cleaner statement of many algorithms and properties of trees.

There are other equivalent ways to define the height of a tree T. One way uses the following definition of the **level** of a node n:

Level of a node

- If n is the root of T, it is at level 1.
- If n is not the root of T, its level is 1 greater than the level of its parent.

For example, in Figure 11-6a, node A is at level 1, node B is at level 2, and node D is at level 3.

The **height** of a tree T in terms of the levels of its nodes is defined as follows:

Height of a tree in terms of levels

- If T is empty, its height is 0.
- If T is not empty, its height is equal to the maximum level of its nodes.

Apply this definition to the trees in Figure 11-6 and show that their heights are, respectively, 3, 5, and 7, as was stated earlier.

For binary trees, it is often convenient to use an equivalent recursive definition of height:

Recursive definition of height

- If T is empty, its height is 0.
- If T is a nonempty binary tree, then because T is of the form

the height of T is 1 greater than the height of its root's taller subtree; that is,

$$height(T) = 1 + max\{height(T_L), height(T_R)\}$$

Later, when we discuss the efficiency of searching a binary search tree, it will be necessary to determine the maximum and minimum heights of a binary tree of n nodes.

Full, complete, and balanced binary trees. In a full binary tree of height h, all nodes that are at a level less than h have two children each. Figure 11-7 depicts a full binary tree of height 3. Each node in a full binary tree has left and right subtrees of the same height. Among binary trees of height h, a full binary tree has as many leaves as possible, and they all are at level h. Intuitively, a full binary tree has no missing nodes.

FIGURE 11-7
A full binary tree of height 3

A full binary tree

When proving properties about full binary trees—such as how many nodes they have—the following recursive definition of a full binary tree is convenient:

- If T is empty, T is a full binary tree of height 0.
- If T is not empty and has height $h > 0$, T is a full binary tree if its root's subtrees are both full binary trees of height $h - 1$.

This definition clearly reflects the recursive nature of a binary tree.

A complete binary tree

A **complete binary tree** of height h is a binary tree that is full down to level $h - 1$, with level h filled in from left to right, as Figure 11-8 illustrates. More formally, a binary tree T of height h is complete if

1. All nodes at level $h - 2$ and above have two children each, and
2. When a node at level $h - 1$ has children, all nodes to its left at the same level have two children each, and
3. When a node at level $h - 1$ has one child, it is a left child.

Full binary trees are complete

Parts 2 and 3 of this definition formalize the requirement that level h be filled in from left to right. Note that a full binary tree is complete.

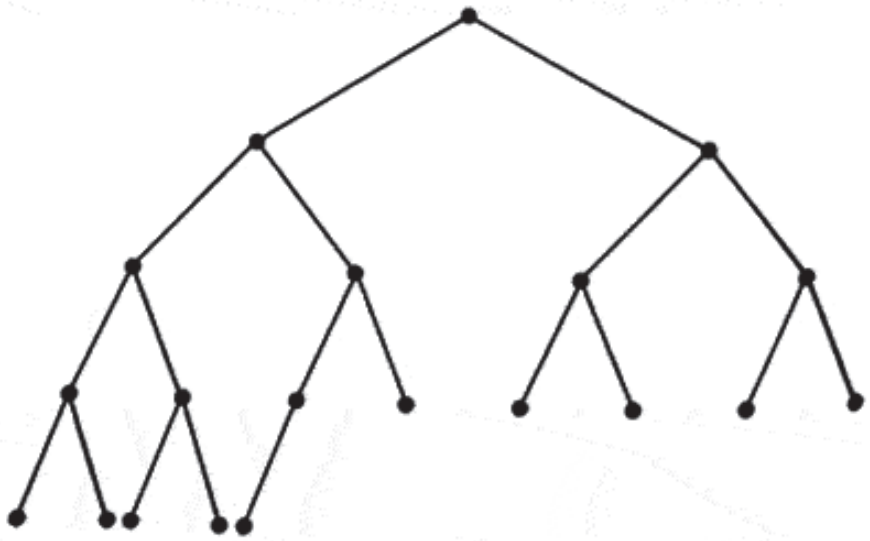


FIGURE 11-8
A complete binary tree

Finally, a binary tree is **height balanced,** or simply **balanced,** if the height of any node's right subtree differs from the height of the node's left subtree by no more than 1. The binary trees in Figures 11-8 and 11-6a are balanced, but the trees in Figures 11-6b and 11–6c are not balanced. A complete binary tree is balanced.

Complete binary trees are balanced

The following is a summary of the major tree terminology presented so far.

KEY CONCEPTS

Summary of Tree Terminology

General tree	A set of one or more nodes, partitioned into a root node and subsets that are general subtrees of the root.
Parent of node *n*	The node directly above node *n* in the tree.
Child of node *n*	A node directly below node *n* in the tree.
Root	The only node in the tree with no parent.
Leaf	A node with no children.
Siblings	Nodes with a common parent.
Ancestor of node *n*	A node on the path from the root to *n*.
Descendant of node *n*	A node on a path from *n* to a leaf.
Subtree of node *n*	A tree that consists of a child (if any) of *n* and the child's descendants.
Height	The number of nodes on the longest path from the root to a leaf.
Binary tree	A set of nodes that is either empty or partitioned into a root node and one or two subsets that are binary subtrees of the root. Each node has at most two children, the left child and the right child.
Left (right) child of node *n*	A node directly below and to the left (right) of node *n* in a binary tree.
Left (right) subtree of node *n*	In a binary tree, the left (right) child (if any) of node *n* plus its descendants.
Binary search tree	A binary tree where the value in any node *n* is greater than the value in every node in *n*'s left subtree, but less than the value of every node in *n*'s right subtree.
Empty binary tree	A binary tree with no nodes.
Full binary tree	A binary tree of height *h* with no missing nodes. All leaves are at level *h* and all other nodes each have two children.

(continues)

KEY CONCEPTS

Summary of Tree Terminology (continued)

Complete binary tree	A binary tree of height h that is full to level $h-1$ and has level h filled in from left to right.
Balanced binary tree	A binary tree in which the left and right subtrees of any node have heights that differ by at most 1.

11.2 The ADT Binary Tree

As an abstract data type, the binary tree has operations that add and remove nodes and subtrees. By using these basic operations, you can build any binary tree. Other operations set or retrieve the data in the root of the tree and determine whether the tree is empty.

Traversal operations that *visit* every node in a binary tree are also typical. "Visiting" a node means "doing something with or to" the node. Chapter 5 introduced the concept of traversal for a linked list: Beginning with the list's first node, you visit each node sequentially until you reach the end of the linked list. Chapter 9 showed how to implement a Java iterator to facilitate traversal of the list. Traversal of a binary tree, however, visits the tree's nodes in one of several different orders. The three standard orders are called preorder, inorder, and postorder, and they are described in the next section along with an iterator for binary trees.

The operations available for a particular ADT binary tree depend on the type of binary tree being implemented. Thus, we will first develop an abstract class representing a binary tree, containing only the most basic binary tree operations. Later, we will extend this abstract class to provide additional binary tree operations.

Basic Operations of the ADT Binary Tree

The first task is to define the operations that are common to all binary tree implementations. Here is a summary:

KEY CONCEPTS

Basic Operations of the ADT Binary Tree

1. Create an empty binary tree.
2. Create a one-node binary tree, given an item.
3. Remove all nodes from a binary tree, leaving it empty.
4. Determine whether a binary tree is empty.
5. Determine what data is the binary tree's root.
6. Set the data in the binary tree's root (may not be implemented by all binary trees).

Notice that we included in this basic set of operations an operation that changes the item in the root. For some binary trees, such an operation would not be desirable, and may not be implemented. The operation should throw an exception to indicate this.

The following pseudocode specifies these basic operations in more detail.

KEY CONCEPTS

Pseudocode for the Basic Operations of the ADT Binary Tree

```
+createBinaryTree()
// Creates an empty binary tree.

+createBinaryTree(in rootItem:TreeItemType)
// Creates a one-node binary tree whose root contains
// rootItem.

+makeEmpty()
// Removes all of the nodes from a binary tree, leaving an
// empty tree.

+isEmpty():boolean {query}
// Determines whether a binary tree is empty.

+getRootItem():TreeItemType throws TreeException {query}
// Retrieves the data item in the root of a nonempty
// binary tree. Throws TreeException if the tree is
// empty.

+setRootItem(in rootItem:TreeItemType)
             throws UnsupportedOperationException
// Sets the data item in the root of a binary tree. Throws
// UnsupportedOperationException if the method is not
// implemented.
```

As you can see, we must still specify other operations for building the tree. One possible set of operations is presented next.

General Operations of the ADT Binary Tree

As was mentioned earlier, the particular operations provided for an ADT binary tree depend on the kind of binary tree we are designing. This section specifies some general operations for a binary tree, with the assumption that we are adding these operations to the basic operations of the ADT binary tree specified before. A UML diagram for a binary tree is shown in Figure 11-9.

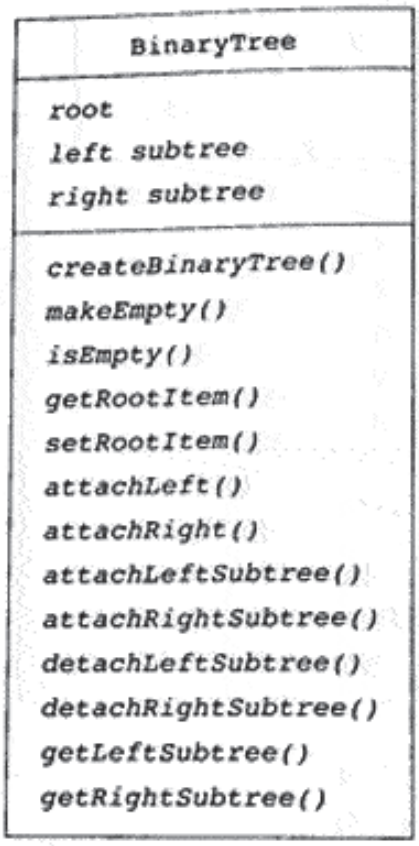


FIGURE 11-9
UML diagram for the class ***BinaryTree***

KEY CONCEPTS

Pseudocode for the General Operations of the ADT Binary Tree

```
+createBinaryTree(in rootItem:TreeItemType,
                  in leftTree:BinaryTree,
                  in rightTree:BinaryTree)
// Creates a binary tree whose root contains rootItem and
// has leftTree and rightTree, respectively, as its left
// and right subtrees.

+setRootItem(in newItem:TreeItemType)
// Replaces the data item in the root of a binary tree
// with newItem, if the tree is not empty. If the
// tree is empty, creates a root node whose data item
// is newItem and inserts the new node into the tree.
```

(continues)

KEY CONCEPTS

Pseudocode for the General Operations of the ADT Binary Tree (continued)

```
+attachLeft(in newItem:TreeItemType) throws TreeException
  Attaches a left child containing newItem to the root of
  a binary tree. Throws TreeException if the binary
  tree is empty (no root node to attach to) or a left
  subtree already exists (should explicitly detach it
  first).

+attachRight(in newItem:TreeItemType) throws TreeException
// Attaches a right child containing newItem to the root of
// a binary tree. Throws TreeException if the binary
// tree is empty (no root node to attach to) or a left
// subtree already exists (should explicitly detach it
// first).

+attachLeftSubtree(in leftTree:BinaryTree) throws TreeException
// Attaches leftTree as the left subtree of the
// root of a binary tree and makes leftTree empty
// so that it cannot be used as a reference into this tree.
// Throws TreeException if the binary tree is empty
// (no root node to attach to) or a left subtree already
// exists (should explicitly detach it first).

+attachRightSubtree(in rightTree:BinaryTree) throws TreeException
// Attaches rightTree as the right subtree of the
// root of a binary tree and makes rightTree empty
// so that it cannot be used as a reference into this tree.
// Throws TreeException if the binary tree is empty
// (no root node to attach to) or a right subtree already
// exists (should explicitly detach it first).

+detachLeftSubtree():BinaryTree throws TreeException
// Detaches and returns the left subtree of a binary tree's
// root. Throws TreeException if the binary tree is empty
// (no root node to detach from).

+detachRightSubtree():BinaryTree throws TreeException
// Detaches and returns the right subtree of a binary tree's
// root. Throws TreeException if the binary tree is empty
// (no root node to detach from).
```

You can use these operations, for example, to build the binary tree in Figure 11-6b, in which the node labels represent character data. The following pseudocode constructs the tree from the subtree `tree1` rooted at `"F"`, the subtree `tree2` rooted at `"D"`, the subtree `tree3` rooted at `"B"`, and the subtree `tree4` rooted at `"C"`. Initially, these subtrees exist but are empty.

Using ADT binary tree operations to build a binary tree

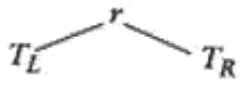
```
tree1.setRootItem("F")
tree1.attachLeft("G")

tree2.setRootItem("D")
tree2.attachLeftSubtree(tree1)

tree3.setRootItem("B")
tree3.attachLeftSubtree(tree2)
tree3.attachRight("E")

tree4.setRootItem("C")
// tree in Fig 11-6b
binTree.createBinaryTree("A", tree3, tree4)
```

The traversal operations are considered in detail next.

Traversals of a Binary Tree

A traversal algorithm for a binary tree visits each node in the tree. While visiting a node, you do something with or to the node. For the purpose of this discussion, assume that visiting a node simply means displaying the data portion of the node.

With the recursive definition of a binary tree in mind, you can construct a recursive traversal algorithm as follows. According to the definition, the binary tree T is either empty or is of the form



If T is empty, the traversal algorithm takes no action—an empty tree is the base case. If T is not empty, the traversal algorithm must perform three tasks: It must display the data in the root r, and it must traverse the two subtrees T_L and T_R, each of which is a binary tree smaller than T.

The general form of a recursive traversal algorithm

Thus, the general form of the recursive traversal algorithm is

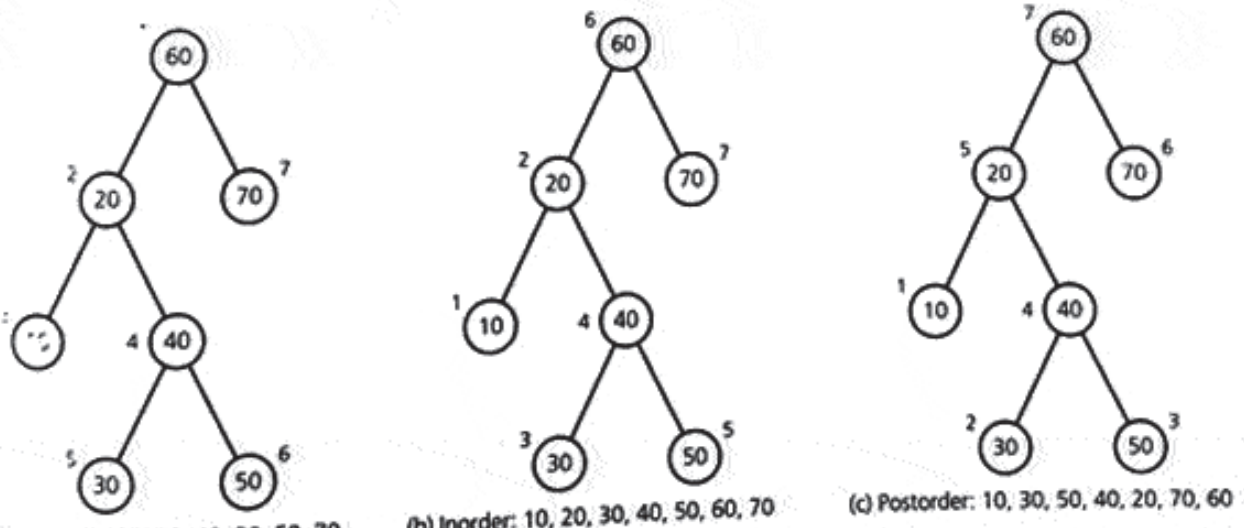
```
+traverse(in binTree:BinaryTree)
// Traverses the binary tree binTree.

 if (binTree is not empty) {
   traverse(Left subtree of binTree's root)
   traverse(Right subtree of binTree's root)
 }  // end if
```

algorithm is not quite complete, however. It is missing the instruction to the data in the root. When traversing any binary tree, the algorithm has choices of when to visit the root r. It can visit r before it traverses both of subtrees, it can visit r after it has traversed r's left subtree T_L but before it ses r's right subtree T_R, or it can visit r after it has traversed both of r's ees. These traversals are called **preorder, inorder,** and **postorder,** respec- Figure 11-10 shows the results of these traversals for a given binary tree. he preorder traversal algorithm is as follows:

Preorder traversal

```
preorder(in binTree:BinaryTree)
  Traverses the binary tree binTree in preorder.
  Assumes that "visit a node" means to display
  the node's data item.

if (binTree is not empty) {
  Display the data in the root of binTree
  preorder(Left subtree of binTree's root)
  preorder(Right subtree of binTree's root)
    end if
```

The preorder traversal of the tree in Figure 11-10a visits the nodes in this order: 60, 20, 10, 40, 30, 50, 70. If you apply preorder traversal to a binary tree that represents an algebraic expression, such as any tree in Figure 11-4,



(a) Preorder: 60, 20, 10, 40, 30, 50, 70 (b) Inorder: 10, 20, 30, 40, 50, 60, 70 (c) Postorder: 10, 30, 50, 40, 20, 70, 60

(Numbers beside nodes indicate traversal order.)

FIGURE 11-10
Traversals of a binary tree: (a) preorder; (b) inorder; (c) postorder

and display the nodes as you visit them, you will obtain the prefix form of the expression.[1]

Inorder traversal

The inorder traversal algorithm is as follows:

```
+inorder(in binTree:BinaryTree)
// Traverses the binary tree binTree in inorder.
// Assumes that "visit a node" means to display
// the node's data item.

  if (binTree is not empty) {
    inorder(Left subtree of binTree's root)
    Display the data in the root of binTree
    inorder(Right subtree of binTree's root)
  }  // end if
```

The result of the inorder traversal of the tree in Figure 11-10b is 10, 20, 30, 40, 50, 60, 70. If you apply inorder traversal to a binary search tree, you will visit the nodes in order according to their data values. Such is the case for the tree in Figure 11-10b.

Postorder traversal

Finally, the postorder traversal algorithm is as follows:

```
+postorder(in binTree:BinaryTree)
// Traverses the binary tree binTree in postorder.
// Assumes that "visit a node" means to display
// the node's data item.

  if (binTree is not empty) {
    postorder(Left subtree of binTree's root)
    postorder(Right subtree of binTree's root)
    Display the data in the root of binTree
  }  // end if
```

The result of the postorder traversal of the tree in Figure 11-10c is 10, 30, 50, 40, 20, 70, 60. If you apply postorder traversal to a binary tree that represents an algebraic expression, such as any tree in Figure 11-4, and display the nodes as you visit them, you will obtain the postfix form of the expression.[2]

Each of these traversals visits every node in a binary tree exactly once. Thus, n visits occur for a tree of n nodes. Each visit performs the same operations on each node, independently of n, so it must be O(1). Thus, each traversal is O(n).

Traversal is O(n)

As we discussed in Chapter 9, an iterator class can be developed in conjunction with a collection of objects. In this case, the objects are stored in the nodes of a binary tree, and the order in which this collection is iterated could

1. The prefix expressions are (a) $-ab$; (b) $-a/bc$; (c) $*-abc$.
2. The postfix expressions are (a) $ab-$; (b) $abc/-$; (c) $ab-c*$.

...ased on preorder, inorder, or postorder traversal of the tree. This allows ...ts of the binary tree class to visit each node of the tree and specify the ...on to be performed on each item in the tree. The implementation details of ... an iterator will be discussed shortly.

Possible Representations of a Binary Tree

...can implement a binary tree by using the constructs of Java in one of three ...eral ways. Two of these approaches use arrays, but the typical implementa... uses references. In each case, the described data structures would be ... ate data fields of a class of binary trees.

To illustrate the three approaches, we will implement a binary tree of ... Each node in this tree contains a name, and, because the tree is a ... tree, each node has at most two descendant nodes.

An array-based representation. If you use a Java class to define a node in ... tree, you can represent the entire binary tree by using an array of tree ... Each tree node contains a data portion—a name in this case—and two ..., one for each of the node's children, as the following Java statements ...

```
public class TreeNode<T> {
  private T item;              // data item in the tree
  private int leftChild;       // index to left child
  private int rightChild;      // index to right child
  ...
  // constructors and methods appear here
}  // end TreeNode

public class BinaryTreeArrayBased<T> {
  protected final int MAX_NODES = 100;
  protected ArrayList<TreeNode<T>> tree;
  protected int root; // index of tree's root
  protected int free; // index of next unused array
                      // location

  ...
  // constructors and methods
}  // end BinaryTreeArrayBased
```

The constants and data fields are declared *protected* in *BinaryTreeArrayBased* so that they will be directly accessible by the subclasses.

The variable *root* is an index to the tree's root within the array *tree*. If the tree is empty, *root* is –1. Both *leftChild* and *rightChild* within a node are indexes to the children of that node. If a node has no left child, *leftChild* is –1; if a node has no right child, *rightChild* is –1.

As the tree changes due to insertions and deletions, its nodes may not be in contiguous elements of the array. Therefore, this implementation requires

A free list keeps track of available nodes

you to establish a list of available nodes, which is called a **free list.** To insert a new node into the tree, you first obtain an available node from the free list. If you delete a node from the tree, you place it into the free list so that you can reuse the node at a later time. The variable `free` is the index to the first node in the free list and, arbitrarily, the `rightChild` field of each node in the free list is the index of the next node in the free list.[3] Figure 11-11 contains a binary tree and the data fields for its array-based implementation.

An array-based representation of a complete tree. The previous implementation works for any binary tree, even though the tree in Figure 11-11 is complete. If you know that your binary tree is complete, you can use a simpler array-based implementation that saves memory. As you saw earlier in this chapter, a complete tree of height h is full to level $h - 1$ and has level h filled from left to right.

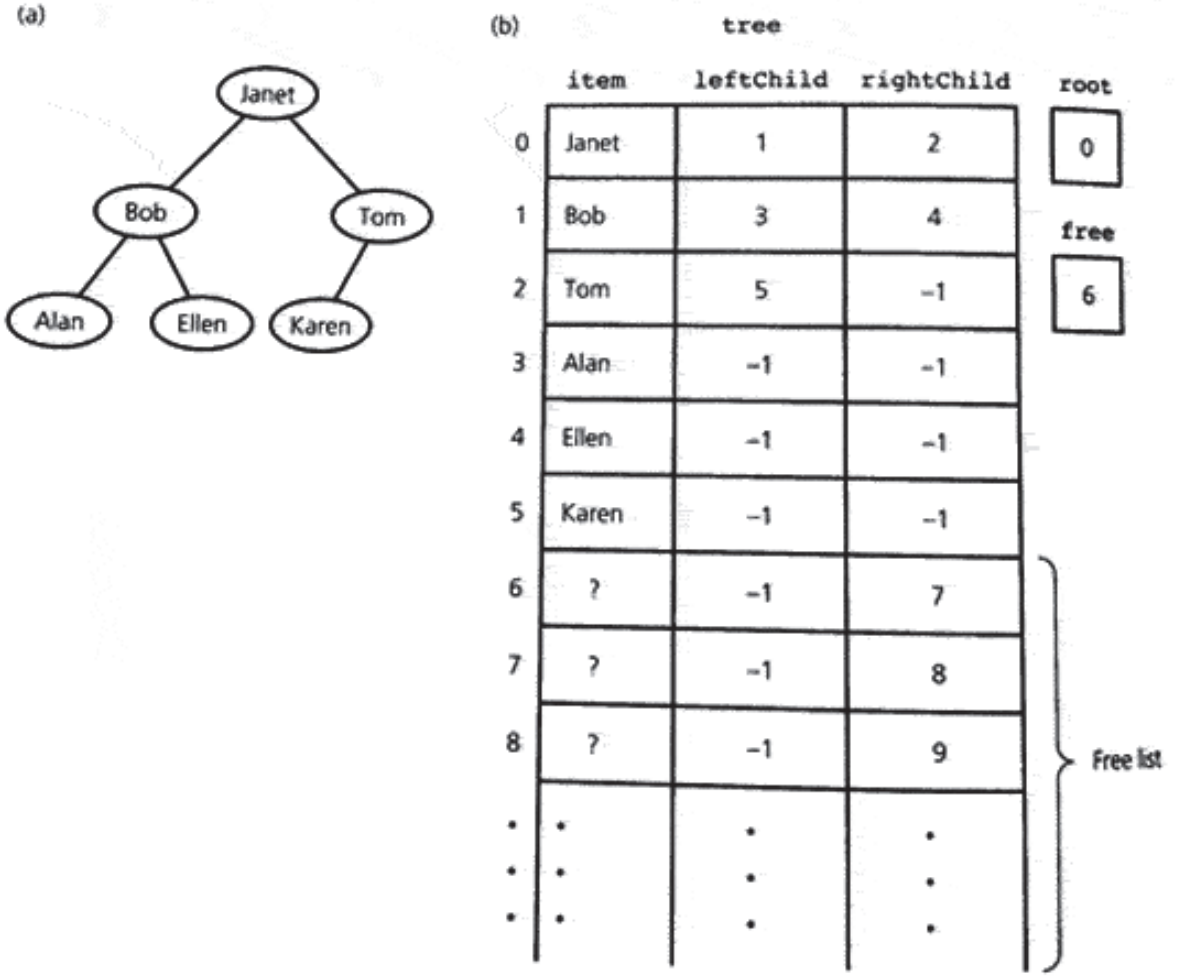


FIGURE 11-11

(a) A binary tree of names; (b) its array-based implementation

3. This free list is actually an array-based linked list, as Programming Problem 10 of Chapter 5 describes.

Figure 11-12 shows the complete binary tree of Figure 11-11a with its nodes numbered according to a standard level-by-level scheme. The root is numbered 0, and the children of the root (the next level of the tree) are numbered, left to right, 1 and 2. The nodes at the next level are numbered, left to right, 3, 4, and 5. You place these nodes into the array `tree` in numeric order. That is, `tree[i]` contains the node numbered i, as Figure 11-13 illustrates. Now, given any node `tree[i]`, you can easily locate both of its children and its parent: Its left child (if it exists) is `tree[2*i+1]`, its right child (if it exists) is `tree[2*i+2]`, and its parent (if `tree[i]` is not the root) is `tree[(i-1)/2]`.

If the binary tree is complete and remains complete you can use a memory-efficient array-based implementation

This array-based representation requires a complete binary tree. If nodes were missing from the middle of the tree, the numbering scheme would be thrown off, and the parent-child relationship among nodes would be ambiguous. This requirement implies that any changes to the tree must maintain its completeness.

As you will see in the next chapter, an array-based representation of a binary tree is useful in the implementation of the ADT priority queue.

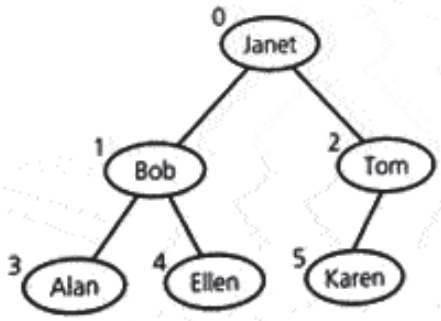


FIGURE 11-12
Level-by-level numbering of a complete binary tree

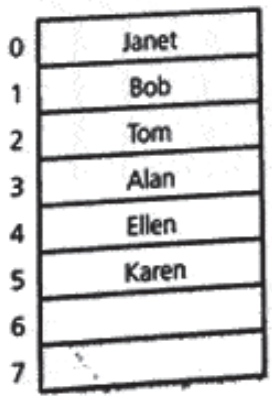


FIGURE 11-13
An array-based implementation of the complete binary tree in Figure 11-12

A reference-based representation. You can use Java references to link the nodes in the tree. Thus, you can represent a tree by using the following Java classes:

```
class TreeNode<T> {
  T item;
  TreeNode<T> leftChild;
  TreeNode<T> rightChild;
  ...
  // constructors
}  // end TreeNode

public abstract class BinaryTreeBasis<T> {
  protected TreeNode root;
  ...
  // constructors and methods appear here
}  // end BinaryTreeBasis
```

The class *TreeNode* is analogous to the class *Node* that we used in Chapter 5 for a linked list. The data fields of *TreeNode* are declared package access only. Within the class *BinaryTreeBasis*, the external reference *root* references the tree's root. If the tree is empty, *root* is *null*. Figure 11-14 illustrates this implementation.

The root of a nonempty binary tree has a left subtree and a right subtree, each of which is a binary tree. In a reference-based implementation, *root* references the root *r* of a binary tree, *root.leftChild* references the root of the left subtree of *r*, and *root.rightChild* references the root of the right subtree of *r*.

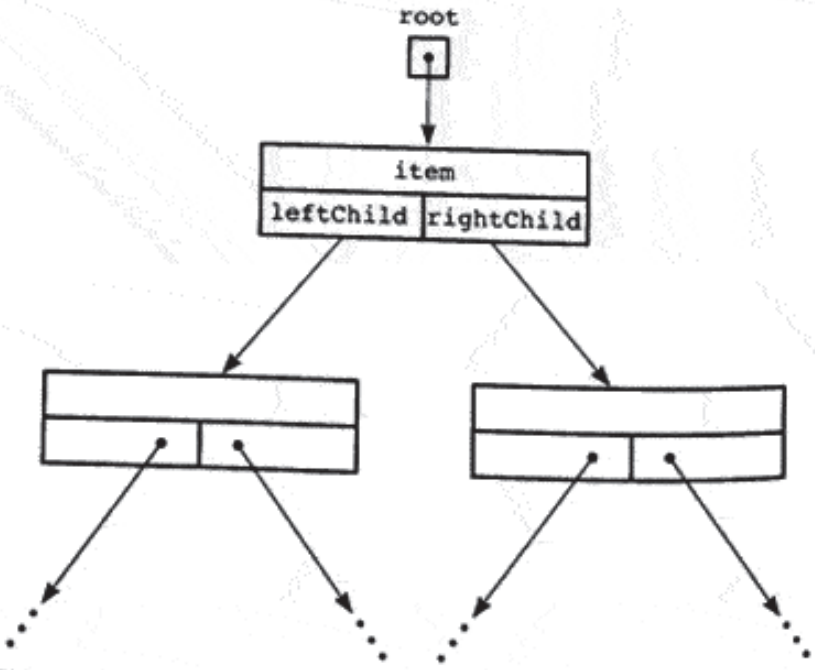


FIGURE 11-14
A reference-based implementation of a binary tree

The section that follows provides the details for a reference-based implementation of the ADT binary tree.

A Reference-Based Implementation of the ADT Binary Tree

The following classes provide a generic reference-based implementation for the ADT binary tree described earlier. A discussion of several implementation details follows these classes. The specification of pre- and postconditions is left as an exercise.

A node in a binary tree

```
class TreeNode<T> {
  T item;
  TreeNode<T> leftChild;
  TreeNode<T> rightChild;

  public TreeNode(T newItem) {
     Initializes tree node with item and no children.
    item = newItem;
    leftChild  = null;
    rightChild = null;
  }  // end constructor

  public TreeNode(T newItem,
                  TreeNode<T> left, TreeNode<T> right) {
    // Initializes tree node with item and
    // the left and right children references.
    item = newItem;
    leftChild  = left;
    rightChild = right;
  }  // end constructor

}  // end TreeNode
```

An exception class

```
public class TreeException extends RuntimeException {
  public TreeException(String s) {
    super(s);
  }  // end constructor
} // end TreeException
```

An abstract class of basic tree operators

```
public abstract class BinaryTreeBasis<T> {
  protected TreeNode<T> root;

  public BinaryTreeBasis() {
    root = null;
  }  // end default constructor

  public BinaryTreeBasis(T rootItem) {
    root = new TreeNode<T>(rootItem, null, null);
  }  // end constructor
```

```
  public boolean isEmpty() {
  // Returns true if the tree is empty, else returns false.
    return root == null;
  }  // end isEmpty

  public void makeEmpty() {
  // Removes all nodes from the tree.
    root = null;
  }  // end makeEmpty

  public T getRootItem() throws TreeException {
  // Returns the item in the tree's root.
    if (root == null) {
      throw new TreeException("TreeException: Empty tree");
    }
    else {
      return root.item;
    }  // end if
  }  // end getRootItem

  public abstract void setRootItem(T newItem);
    // Throws UnsupportedOperationException if operation
    // is not supported.

}  // end BinaryTreeBasis
```

BinaryTreeBasis will be used as the base class for the implementation of particular binary trees. It is declared as an abstract class because it is used for inheritance purposes only; there can be no direct instances of this class. ***BinaryTreeBasis*** declares the root of the tree as a protected item so that the subclasses will have direct access to the root of the tree. It also provides methods to check for an empty tree, to make the tree empty, and to retrieve the contents of the root node. Setting the contents of the root node is left to a subclass, since different kinds of binary trees may have varying requirements regarding the value in the root node of the tree. Some implementations may not support the operation ***setRootItem***, they should be implemented to throw ***UnsupportedOperationException***.

A class that extends **BinaryTreeBasis <>**

The following class implements the general operations of a binary tree and is derived from ***BinaryTreeBasis***.

```
public class BinaryTree<T> extends BinaryTreeBasis<T> {

  public BinaryTree() {
  }  // end default constructor
```

```
public BinaryTree(T rootItem) {
  super(rootItem);
} // end constructor

public BinaryTree(T rootItem,
                  BinaryTree<T> leftTree,
                  BinaryTree<T> rightTree) {
  root = new TreeNode<T>(rootItem, null, null);
  attachLeftSubtree(leftTree);
  attachRightSubtree(rightTree);
} // end constructor

public void setRootItem(T newItem) {
  if (root != null) {
    root.item = newItem;
  }
  else {
    root = new TreeNode<T>(newItem, null, null);
  } // end if
} // end setRootItem

public void attachLeft(T newItem) {
  if (!isEmpty() && root.leftChild == null) {
    // assertion: nonempty tree; no left child
    root.leftChild = new TreeNode<T>(newItem, null, null);
  } // end if
} // end attachLeft

public void attachRight(T newItem) {
  if (!isEmpty() && root.rightChild == null) {
    // assertion: nonempty tree; no right child
    root.rightChild = new TreeNode<T>(newItem, null, null);
  } // end if
} // end attachRight

public void attachLeftSubtree(BinaryTree<T> leftTree)
                              throws TreeException {
  if (isEmpty()) {
    throw new TreeException("TreeException:  Empty tree");
  }
  else if (root.leftChild != null) {
    // a left subtree already exists; it should have been
    // deleted first
    throw new TreeException("TreeException: " +
                            "Cannot overwrite left subtree");
  }
```

```
    else {
      // assertion: nonempty tree; no left child
      root.leftChild = leftTree.root;
      // don't want to leave multiple entry points into
      // our tree
      leftTree.makeEmpty();
    } // end if
  } // end attachLeftSubtree

  public void attachRightSubtree(BinaryTree<T> rightTree)
                                        throws TreeException {
    if (isEmpty()) {
      throw new TreeException("TreeException:  Empty tree");
    }
    else if (root.rightChild != null) {
      // a right subtree already exists; it should have been
      // deleted first
      throw new TreeException("TreeException: " +
                              "Cannot overwrite right subtree");
    }
    else {
      // assertion: nonempty tree; no right child
      root.rightChild = rightTree.root;
      // don't want to leave multiple entry points into
      // our tree
      rightTree.makeEmpty();
    } // end if
  } // end attachRightSubtree

  protected BinaryTree(TreeNode<T> rootNode) {
    root = rootNode;
  } // end protected constructor

  public BinaryTree<T> detachLeftSubtree()
                              throws TreeException {
    if (isEmpty()) {
      throw new TreeException("TreeException:  Empty tree");
    }
    else {
      // create a new binary tree that has root's left
      // node as its root
      BinaryTree<T> leftTree;
      leftTree = new BinaryTree<T>(root.leftChild);
      root.leftChild = null;
      return leftTree;
    } // end if
  } // end detachLeftSubtree
```

```
  public BinaryTree<T> detachRightSubtree()
                       throws TreeException {
    if (isEmpty()) {
      throw new TreeException("TreeException:  Empty tree");
    }
    else {
      BinaryTree <T> rightTree;
      rightTree = new BinaryTree<T>(root.rightChild);
      root.rightChild = null;
      return rightTree;
    }  // end if
  }  // end detachRightSubtree
} // end BinaryTree
```

The class *BinaryTree* has more constructors than previous classes you have seen. They allow you to define binary trees in a variety of circumstances. Two of these public constructors refer back to the abstract class *BinaryTreeBasis*, with a third public constructor implemented within *BinaryTree*. With these constructors you can construct a binary tree

- That is empty
- From data for its root, which is its only node
- From data for its root and the root's two subtrees

For example, the following statements invoke these three constructors:

Sample uses of public constructors

```
BinaryTree<Integer> tree1 = new BinaryTree<Integer>();
BinaryTree<Integer> tree2 = new BinaryTree<Integer>(root2);
BinaryTree<Integer> tree3 = new BinaryTree<Integer>(root3);
BinaryTree<Integer> tree4 = new BinaryTree<Integer>(root4,
                                                    tree2, tree3);
```

In these statements, *tree1* is an empty binary tree; *tree2* and *tree3* have only root nodes, whose data is *root2* and *root3*, respectively; and *tree4* is a binary tree whose root contains *root4* and has subtrees *tree2* and *tree3*.

The class also contains a protected constructor, which creates a tree from a reference to a root node. For example,

```
leftTree = new BinaryTree<T>(root.leftChild);
```

constructs a tree *leftTree* whose root node th
ences. Although the method detach
use this constructor, it should not be a
do not have access to no

Note that attach
with the subtree as an ar

tree. This causes the root of the subtree as it existed in the client to be set to `null`. Thus, the client will not be able to access and manipulate the subtree directly, a violation of abstraction of the tree.

Tree Traversals Using an Iterator

We will use the tree traversals to determine the order in which an iterator will visit the nodes of a tree. The tree iterator will implement the Java `Iterator` interface and will provide methods to set the iterator to the type of traversal desired. Since the abstract class `BinaryTreeBasis` has sufficient information to perform a traversal, we will define the iterator using this class. This will allow the iterator class to be used by any subclass of the `BinaryTreeBasis`.

As we mentioned in Chapter 9, a class that implements the `Iterator` interface must provide three methods: `next()`, `hasNext()`, and `remove()`. We will not implement the `remove()` method in this version of the iterator for two reasons: First, the semantics of removing a node from a tree may depend on the type of tree you are working with (for example, a binary search tree). Second, the class `BinaryTree` itself does not provide a method for removing nodes from the tree.

The iterator class for the binary tree should be placed in the same package as `BinaryTreeBasis`. Doing so will give the iterator access to the root of the tree and, in turn, to all of the nodes of the tree. This access is necessary for the implementation of the iterator.

You must implement the recursive traversal operations carefully so that you do not violate the wall of the ADT. For example, the method `inorder`, whose declaration is

```
void inorder(TreeNode treeNode);
```

has as a parameter the reference `treeNode`, which eventually references every node in the tree. Because this parameter clearly depends on the tree's reference-based implementation, `inorder` is not suitable as a public method. The method `inorder`, in fact, is a private method, which the public method `setInorder` calls.

Implement traversals so that the action to be performed remains on the client's side of the wall

The implementation presented here for the iterator of a binary tree assumes that the iteration order will be set by calling `setPreorder`, `setInorder`, or `setPostorder`. Until one of these methods is called, the iterator will not provide any items from the tree (`hasNext` returns `false`). Recall from Chapter 9 that the behavior of an iterator is unspecified if the underlying collection is modified in any way other than by calling the method `remove()` while the iteration is in progress. Thus, if the binary tree is altered after the iteration order has been set, the changes to the tree will not be reflected in the iteration.

Here is the definition of the tree iterator class `TreeIterator`:

```
import java.util.LinkedList;
public class TreeIterator<T> implements java.util.Iterator<T> {
  private BinaryTreeBasis<T> binTree;
  private TreeNode<T> currentNode;
  private LinkedList <TreeNode<T>> queue; // from JCF

  public TreeIterator(BinaryTreeBasis<T> bTree) {
    binTree = bTree;
    currentNode = null;
    // empty queue indicates no traversal type currently
    // selected or end of current traversal has been reached
    queue = new LinkedList <TreeNode<T>>();
  }  // end constructor

  public boolean hasNext() {
    return !queue.isEmpty();
  }  // end hasNext

  public T next()
           throws java.util.NoSuchElementException {
      currentNode = queue.remove();
      return currentNode.item;
  }  // end next

  public void remove()
              throws UnsupportedOperationException {
    throw new UnsupportedOperationException();
  }  // end remove

  public void setPreorder() {
    queue.clear();
    preorder(binTree.root);
  }  // setPreOrder

  public void setInorder() {
    queue.clear();
    inorder(binTree.root);
  }  // end setInorder

  public void setPostorder() {
    queue.clear();
    postorder(binTree.root);
  }  // end setPostorder
```

```
    private void preorder(TreeNode<T> treeNode) {
      if (treeNode != null) {
        queue.add(treeNode);
        preorder(treeNode.leftChild);
        preorder(treeNode.rightChild);
      } // end if
    }  // end preorder

    private void inorder(TreeNode<T> treeNode) {
      if (treeNode != null) {
        inorder(treeNode.leftChild);
        queue.add(treeNode);
        inorder(treeNode.rightChild);
      } // end if
    }  // end inorder

    private void postorder(TreeNode<T> treeNode) {
      if (treeNode != null) {
        postorder(treeNode.leftChild);
        postorder(treeNode.rightChild);
        queue.add(treeNode);
      } // end if
    }  // end postorder
  }  // end TreeIterator
```

The class *TreeIterator* uses a queue (using the *LinkedList* class from the JCF) to maintain the current traversal of the nodes in the tree. This traversal order is placed in a queue when the client selects the desired traversal method. If a new traversal is set in the middle of an iteration, the queue is cleared first, and then the new traversal is generated.

The following statements create an iterator that will perform a preorder traversal of a tree *tree4*:

```
TreeIterator<T> treeIterator = new TreeIterator<T>(tree4);
treeIterator.setPreorder();
```

Here is an example that uses the iterator to print out the nodes of the tree using the preorder traversal:

```
System.out.println("Preorder traversal:");
while (treeIterator.hasNext()) {
  System.out.println(treeIterator.next());
}  // end while
```

To demonstrate how to use *BinaryTree* and *TreeIterator*, we build and then traverse the binary tree in Figure 11-10:

A sample program

```
public static void main(String[] args) {
  BinaryTree<Integer> tree3 = new BinaryTree<Integer>(70);

  // build the tree in Figure 11-10
  BinaryTree<Integer> tree1 = new BinaryTree<Integer>();
  tree1.setRootItem(40);
  tree1.attachLeft(30);
  tree1.attachRight(50);

  BinaryTree<Integer> tree2 = new BinaryTree<Integer>();
  tree2.setRootItem(20);
  tree2.attachLeft(10);
  tree2.attachRightSubtree(tree1);

  BinaryTree<Integer> binTree =   // tree in Figure 11-10
        new BinaryTree<Integer>(60, tree2, tree3);

  TreeIterator<Integer> btIterator =
        new TreeIterator<Integer>(binTree);
  btIterator.setInorder();

  while (btIterator.hasNext()) {
    System.out.println(btIterator.next());
  }  // end while

  BinaryTree<Integer> leftTree = binTree.detachLeftSubtree();
  TreeIterator<Integer> leftIterator =
                        new TreeIterator<Integer>(leftTree);

  // iterate through the left subtree
  leftIterator.setInorder();
  while (leftIterator.hasNext()) {
    System.out.println(leftIterator.next());
  }  // end while

  // iterate through binTree minus left subtree
  btIterator.setInorder();
  while (btIterator.hasNext()) {
    System.out.println(btIterator.next());
  }  // end while
} // end main
```

Here, *binTree* is the tree in Figure 11-10. Its inorder traversal is 10, 20, 30, 40, 50, 60, 70. The inorder traversal of the left subtree of *binTree*'s root (the

subtree rooted at 20) is 10, 20, 30, 40, 50. The inorder traversal of `leftTree` produces the same result. Since `leftTree` is actually detached from `binTree`, the final traversal of `binTree` is 60, 70.

One disadvantage of this implementation of the traversal is that it performs a lot of computations that may never be used. Not only is a queue of node references created, but also the recursion stores activation records on an implicit stack. Besides the time requirement, $O(n)$ additional space is used by the recursion for a tree with n nodes. In contrast, the space requirement for a nonrecursive traversal of the tree would be only $O(\textit{height of the tree})$. Programming Problem 6 asks you to explore this issue further.

Nonrecursive traversal (*optional*). Before leaving the topic of traversals, let's develop a nonrecursive traversal algorithm to illustrate further the relationship between stacks and recursion that was discussed in Chapter 7. In particular, we will develop a nonrecursive inorder traversal for the reference-based implementation of a binary tree.

The conceptually difficult part of a nonrecursive traversal is determining where to go next after a particular node has been visited. To gain some insight into this problem, consider how the recursive `inorder` method works:

```
+inorder(in treeNode:TreeNode)
// Recursively traverses a binary tree in inorder.

  if (treeNode != null) {
    inorder(treeNode.leftChild)  // point 1
    queue.enqueue(treeNode)
    inorder(treeNode.rightChild) // point 2
   } // end if
```

Recursive calls from points 1 and 2

The method has its recursive calls marked as points 1 and 2.

During the course of the method's execution, the value of the reference `treeNode` actually marks the current position in the tree. Each time `inorder` makes a recursive call, the traversal moves to another node. In terms of the stack that is implicit to recursive methods, a call to `inorder` pushes the new value of `treeNode`—that is, a reference to the new current node—onto the stack. At any given time, the stack contains references to the nodes along the path from the tree's root to the current node n, with the reference to n at the top of the stack and the reference to the root at the bottom. Note that n is possibly "empty"—that is, it may be indicated by a `null` value for `treeNode` at the top of the stack.

Study recursive **`inorder`**'s implicit stack to gain insight into a nonrecursive traversal algorithm

Figure 11-15 partially traces the execution of `inorder` and shows the contents of the implicit stack. The first four steps of the trace show the stack as `treeNode` references first 60, then 20, then 10, and then becomes `null`. The recursive calls for these four steps are from point 1 in `inorder`.

Now consider what happens when `inorder` returns from a recursive call. The traversal retraces its steps by backing up the tree from a node n to its

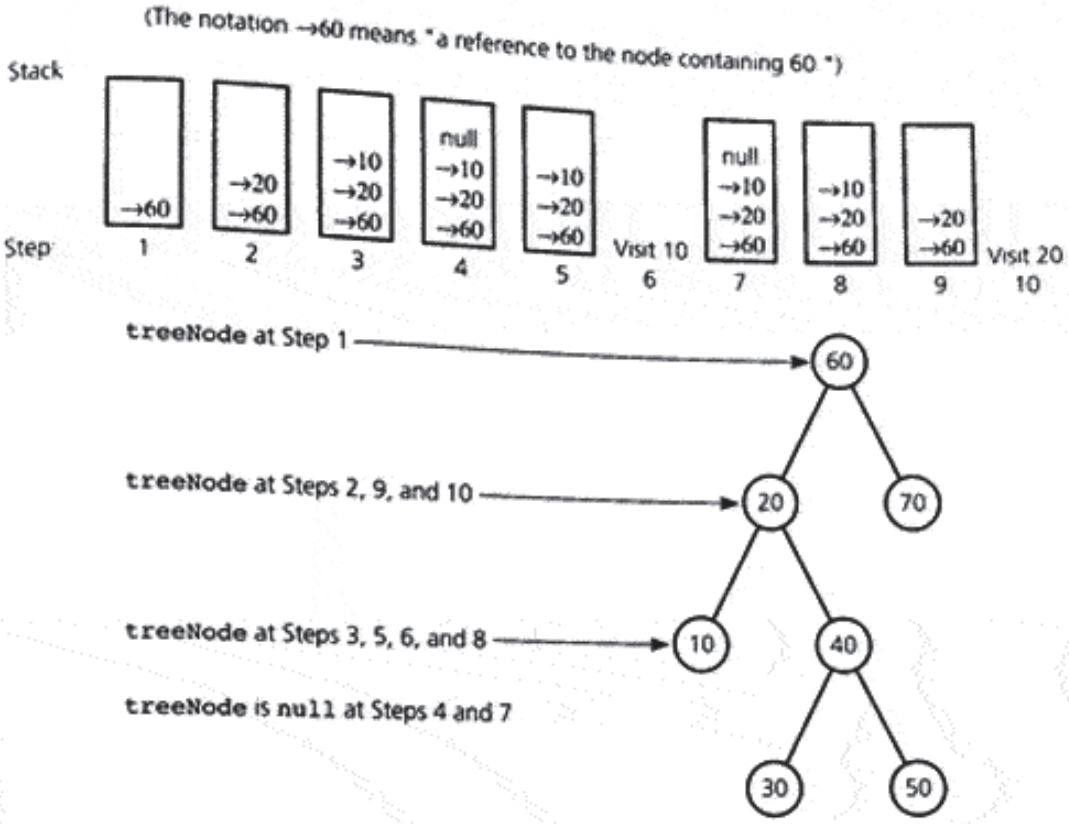


FIGURE 11-15

Contents of the implicit stack as **`treeNode`** progresses through a given tree during a recursive inorder traversal

parent *p*, from which the recursive call to *n* was made. Thus, the reference to *n* is popped from the stack and the reference to *p* comes to the top of the stack, as occurs in Step 5 of the trace in Figure 11-15. (*n* happens to be empty in this case, so *null* is popped from the stack.)

What happens next depends on which subtree of *p* has just been traversed. If you have just finished traversing *p*'s left subtree (that is, if *n* is the left child of *p* and thus the return is made to point 1 in `inorder`), control is returned to the statement that displays the data in node *p*. Such is the case for Steps 6 and 10 of the trace in Figure 11-15. Figure 11-16a illustrates Steps 9 and 10 in more detail.

After the data in *p* has been displayed, a recursive call is made from point 2 and the right subtree of *p* is traversed. However, if, as Figure 11-16b illustrates, you have just traversed *p*'s right subtree (that is, if *n* is the right child of *p* and thus the return is made to point 2), control is returned to the end of the method. As a consequence, another return is made, the reference to *p* is popped off the stack, and you go back up the tree to *p*'s parent, from which the recursive call to *p* was made. In this latter case, the data in *p* is not displayed—it was displayed before the recursive call to *n* was made from point 2.

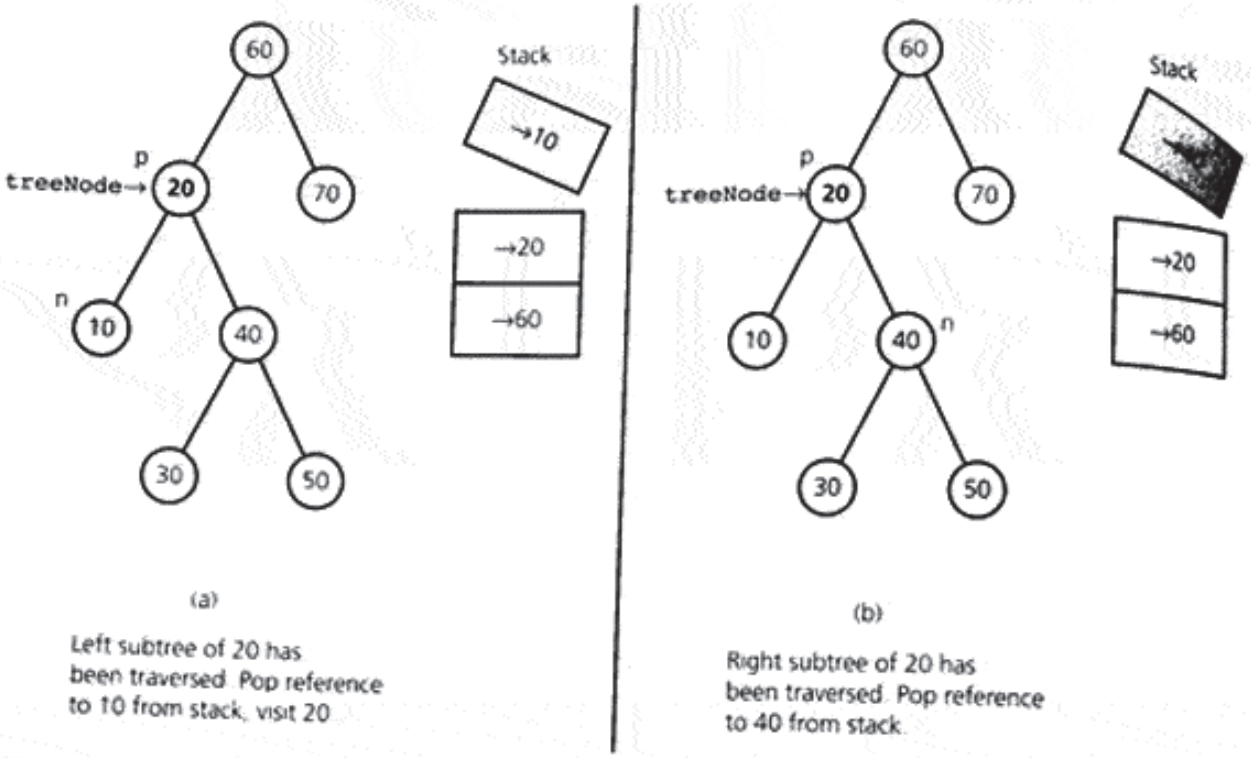


FIGURE 11-16
Traversing (a) the left and (b) the right subtrees of 20

Actions at a return from a recursive call to ***inorder***

Thus, two facts emerge from the recursive version of ***inorder*** when a return is made from a recursive call:

- The implicit recursive stack of references is used to find the node p to which the traversal must go back.
- Once the traversal backs up to node p, it either visits p (for example, displays its data) or backs farther up the tree. It visits p if p's left subtree has just been traversed; it backs up if its right subtree has just been traversed. The appropriate action is taken simply as a consequence of the point—1 or 2—to which control is returned.

You could directly mimic this action by using an iterative method and an explicit stack, as long as some bookkeeping device kept track of which subtree of a node had just been traversed. However, you can use the following observation both to eliminate the need for the bookkeeping device and to speed up the traversal somewhat. Consider the tree in Figure 11-17. After you have finished traversing the subtree rooted at node R, there is no need to return to nodes C and B, because the right subtrees of these nodes have already been traversed. You can instead return directly to node A, which is the nearest ancestor of R whose right subtree has not yet been traversed.

This strategy of not returning to a node after its right subtree has been traversed is simple to implement: You place a reference to a node in the stack only before the node's left subtree is traversed, but not before its right subtree is

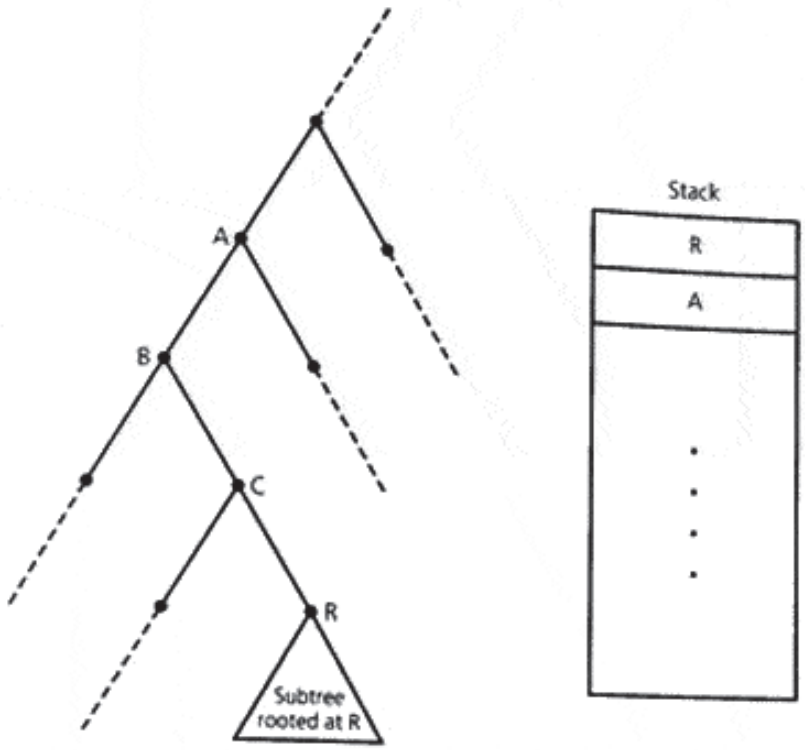


FIGURE 11-17

Returning returns to nodes *B* and *C*

traversed. Thus, in Figure 11-17, when you are at node *R*, the stack contains *A* and *R*, with *R* on top. Nodes *B* and *C* are not in the stack, because you have visited them already and are currently traversing their right subtrees. On the other hand, *A* is in the stack because you are currently traversing its left subtree. When you return from node *R*, nodes *B* and *C* are thus bypassed because you have finished with their right subtrees and do not need to return to these nodes. Thus, you pop *R*'s reference from the stack and go directly to node *A*, whose left subtree has just been traversed. You then visit *A*, pop its reference from the stack, and traverse *A*'s right subtree.

Nonrecursive inorder traversal

This nonrecursive traversal strategy is captured by then following pseudocode, assuming a reference-based implementation. Exercise 17 at the end of this chapter asks you to trace this algorithm for the tree in Figure 11-15.

```
+inorderTraverse(in treeNode:TreeNode)
// Nonrecursively traverses a binary tree in inorder.

 // initialize
 Create an empty stack visitStack
 curr = treeNode                  // start at root treeNode
 done = false
```

```
while (!done) {
  if (curr != null) {
    // place reference to node on stack before
    // traversing node's left subtree
    visitStack.push(curr)

    // traverse the left subtree
    curr = curr.leftChild
  }

  else {// backtrack from the empty subtree and visit
       // the node at the top of the stack; however,
       // if the stack is empty, you are done
    if (!visitStack.isEmpty()) {
      curr = visitStack.pop()
      queue.enqueue(curr)

      // traverse the right subtree
      // of the node just visited
      curr = curr.rightChild
    }

    else {
      done = true
    } // end if
  } // end if
} // end while
```

Eliminating recursion can be more complicated than the example given here. However, the general case is beyond the scope of this book.

11.3 The ADT Binary Search Tree

Searching for a particular item is one operation for which the ADT binary tree is ill suited. The binary search tree is a binary tree that corrects this deficiency by organizing its data by value. Recall that each node n in a binary search tree satisfies the following three properties:

- n's value is greater than all values in its left subtree T_L.
- n's value is less than all values in its right subtree T_R.
- Both T_L and T_R are binary search trees.

This organization of data enables you to search a binary search tree for a particular data item, given its value instead of its position. As you will see, certain conditions make this search efficient.

A binary search tree is often used in situations where the instances stored in the tree contain many different fields of information. For example, each item in a binary search tree might contain a person's name, ID number, address, telephone number, and so on. In general, such an item is called a record and will be an instance of a Java class. To determine whether a particular person is in the tree, you could provide the data for all components, or fields, of the record, but typically you would provide only one field—the ID number, for example. Thus, the request

```
Find the record for the person whose ID number is 123456789
```

is feasible if the ID number uniquely describes the person. By making this request, not only can you determine whether a person is in a binary search tree, but, once you find the person's record, you can also access the other data about the person.

A data item in a binary search tree has a specially designated search key

A field such as an ID number is called a **search key**, or simply a **key**, because it identifies the record that you seek. This portion of the record may involve more than one field, and it will need to be compared to the key of other records. It is important that the value of the search key remain the same as long as the item is stored in the tree. Changing the search key of an existing element in the tree could make that element or other tree elements impossible to find. Thus, the search-key value should not be modifiable. Also, this search-key data type (or one of its superclasses) should implement the `Comparable` interface. This suggests the use of a `KeyedItem` class for items of the tree. The class `KeyedItem` will contain the search key as a data field and a method for accessing the search key. The class is declared abstract since it is only used to derive other classes and appears as follows:

```
public abstract class KeyedItem<KT extends
                                Comparable<? super KT>> {
   private KT searchKey;

   public KeyedItem(KT key) {
      searchKey = key;
   }  // end constructor

   public KT getKey() {
      return searchKey;
   } // end getKey
} // end KeyedItem
```

Classes for the items that are in a binary search tree must extend `KeyedItem`. Such classes will have only the constructor available for initializing the search key. Thus, the search-key value cannot be modified once an item is created.

As an example of a class that extends `KeyedItem`, suppose we want to create a class `Person` as just described, in which the person's ID number is used as the search key.

```
public class Person extends KeyedItem<String> {
  // inherits method getKey that returns the search key
  private FullName name;
  private String phoneNumber;
  private Address address;

  public Person(String id, FullName name, String phone,
                Address addr) {
    super(id);   // sets the key value to String id
    this.name = name;
    phoneNumber = phone;
    address = addr;
  }  // end constructor

  public String toString() {
    return getKey() + " # " + name;
  } // end toString

  // other methods would appear here
}  // end Person
```

Notice that the class *Person* inherits the *getKey* method to return the field that has been designated as the key. This will be the value that is used to search for a record in the tree. The *Person* class could easily be designed to use a key that involves more than one value. For example, if the key is the person's first name and last name, you can use the class *FullName* as presented on page 223–224 of Chapter 4, since it implements the *Comparable* interface. The class *Person* would then provide the person's full name as the key by replacing the generic parameter with *Fullname* to yield the following:

```
public class Person extends KeyedItem <Fullname> {
  private String idNumber;
  private String phoneNumber;
  private Address address;

  public Person(String id, Fullname name,
                String phone, Address addr) {
    super(name);
    idNumber = id;
    phoneNumber = phone;
    address = addr;
  } // end constructor
  ...
  // other methods appear here
```

The data fields of *Person* have been revised by omitting *name* and adding *idNumber*.

For simplicity, we will assume that the search key uniquely identifies the records in your binary search tree. In this case, you can restate the recursive definition of a binary search tree as follows:

A recursive definition of a binary search tree

For each node n, a binary search tree satisfies the following three properties:

n's search key is greater than all search keys in n's left subtree T_L.

n's search key is less than all search keys in n's right subtree T_R.

Both T_L and T_R are binary search trees.

As an ADT, the binary search tree has operations that are like the operations for the ADTs you studied in previous chapters in that they involve inserting, deleting, and retrieving data. In the implementations of the position-oriented ADTs list, stack, and queue, insertion and deletion into the ADT was independent of the value of the data. In the binary search tree, however, the insertion, deletion, and retrieval operations are by search-key value, not by position. The search key facilitates the search process, since we need to know only the key information to find a particular record or to determine the proper position for a data item in the tree. The traversal operations that you just saw for a binary tree apply to a binary search tree without change, because a binary search tree is a binary tree.

The operations that extend the basic ADT binary tree to the ADT binary search tree are as follows:

KEY CONCEPTS

Operations of the ADT Binary Search Tree

1. Insert a new item into a binary search tree.
2. Delete the item with a given search key from a binary search tree.
3. Retrieve the item with a given search key from a binary search tree.
4. Traverse the items in a binary search tree in preorder, inorder, or postorder.

The UML diagram in Figure 11-18 shows the basic tree operations in the class `BinaryTreeBasis`, which is then used to derive the classes `BinaryTree` and `BinarySearchTree`. The `TreeIterator` is based upon the class `BinaryTreeBasis`, and can be used by instances of both `BinaryTree` and `BinarySearchTree`.

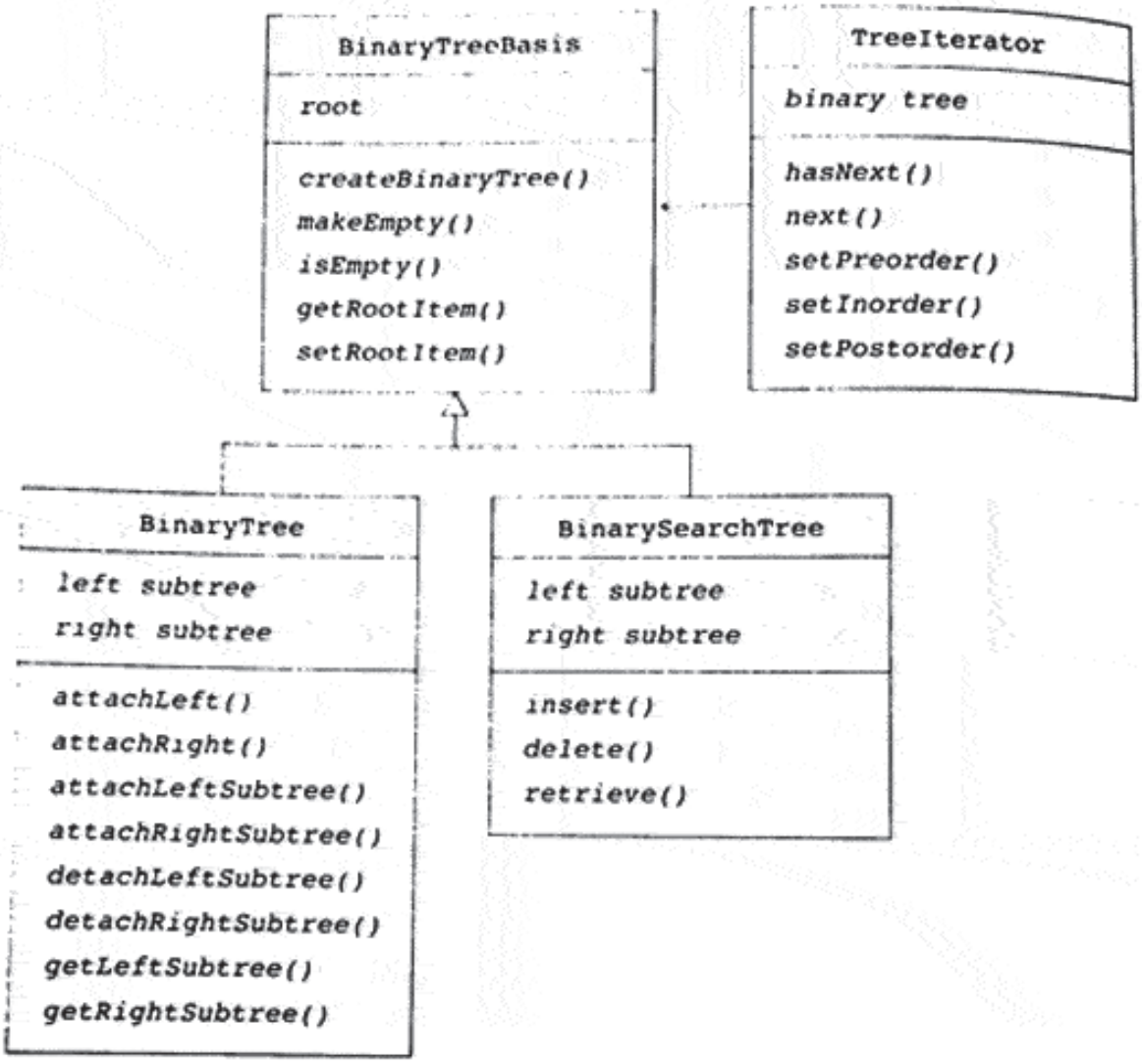


FIGURE 11-18
UML diagram for the binary tree implementations

The following pseudocode specifies the first three operations in more detail. As you soon will see, you can use the class `TreeIterator` developed earlier to perform traversals of a binary search tree.

KEY CONCEPTS

Pseudocode for the Operations of the ADT Binary Search Tree

```
+insert(in newItem:TreeItemType)
// Inserts newItem into a binary search tree whose items
// have distinct search keys that differ from newItem's
// search key.

+delete(in searchKey:KeyType) throws TreeException
// Deletes from a binary search tree the item whose search
// key equals searchKey. If no such item exists, the
// operation fails and throws TreeException.

+retrieve(in searchKey:KeyType):TreeItemType
// Returns the item in a binary search
// tree whose search key equals searchKey. Returns
// null if no such item exists.
```

Figure 11-19 is a binary search tree *nameTree* of names. Each node in the tree is actually a record that represents the named person. That is, if the search key for each record is the person's name, you see only the search keys in the picture of the tree.

For example,

```
firstRecord = nameTree.retrieve("Karen")
```

retrieves Karen's record into *firstRecord*. If you insert a record *secondRecord* into *nameTree* that describes Sarah by invoking

```
nameTree.insert(secondRecord)
```

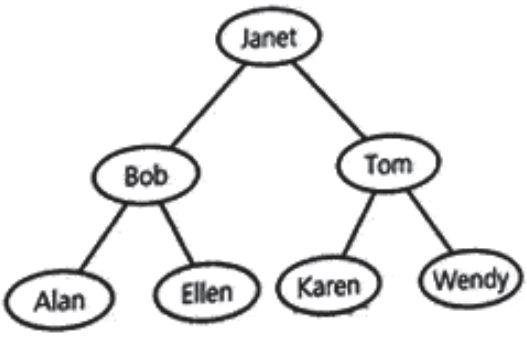


FIGURE 11-19
A binary search tree

you will be able to retrieve that record later and still be able to retrieve Karen's record. If you delete Janet's record by using

```
nameTree.delete("Janet")
```

you will still be able to retrieve the records for Karen and Sarah. Finally, if an iterator *nameIterator* is declared for *nameTree*, you display the name records as follows:

```
TreeIterator<Person> nameIterator =
                                  new TreeIterator<Person>(nameTree);
nameIterator.setInorder();
System.out.println("Inorder traversal:");
while (nameIterator.hasNext()) {
  System.out.println(nameIterator.next());
}  // end while
```

This will display in alphabetical order the names of the people that *nameTree* represents.

Algorithms for the Operations of the ADT Binary Search Tree

Consider again the binary search tree in Figure 11-19. Each node in the tree contains data for a particular person. The person's first name is the search key, and that is the only data item you see in the figure.

Because a binary search tree is recursive by nature, it is natural to formulate recursive algorithms for operations on the tree. Suppose that you want to locate Ellen's record in the binary search tree of Figure 11-19. Janet is in the root node of the tree, so if Ellen's record is present in the tree it must be in Janet's left subtree, because the search key Ellen is before the search key Janet alphabetically. From the recursive definition, you know that Janet's left subtree is also a binary search tree, so you use exactly the same strategy to search this subtree for Ellen. Bob is in the root of this binary search tree, and, because the search key Ellen is greater than the search key Bob, Ellen's record must be in Bob's right subtree. Bob's right subtree is also a binary search tree, and it happens that Ellen is in the root node of this tree. Thus, the search has located Ellen's record.

A search algorithm for a binary search tree

The following pseudocode summarizes this search strategy:

```
+search(in bst:BinarySearchTree, in searchKey:KeyType)
// Searches the binary search tree bst for the item
// whose search key is searchKey.

 if (bst is empty) {
   The desired record is not found
 }
```

```
else if (searchKey == search key of root's item) {
    The desired record is found
}
else if (searchKey < search key of root's item) {
    search(Left subtree of bst, searchKey)
}
else {
  search(Right subtree of bst, searchKey)
}  // end if
```

As you will see, this *search* algorithm is the basis of the insertion, deletion, and retrieval operations on a binary search tree.

Many different binary search trees can contain the names Alan, Bob, Ellen, Janet, Karen, Tom, and Wendy. For example, in addition to the tree in Figure 11-19, each tree in Figure 11-20 is a valid binary search tree for these names. Although

Several different binary search trees are possible for the same data

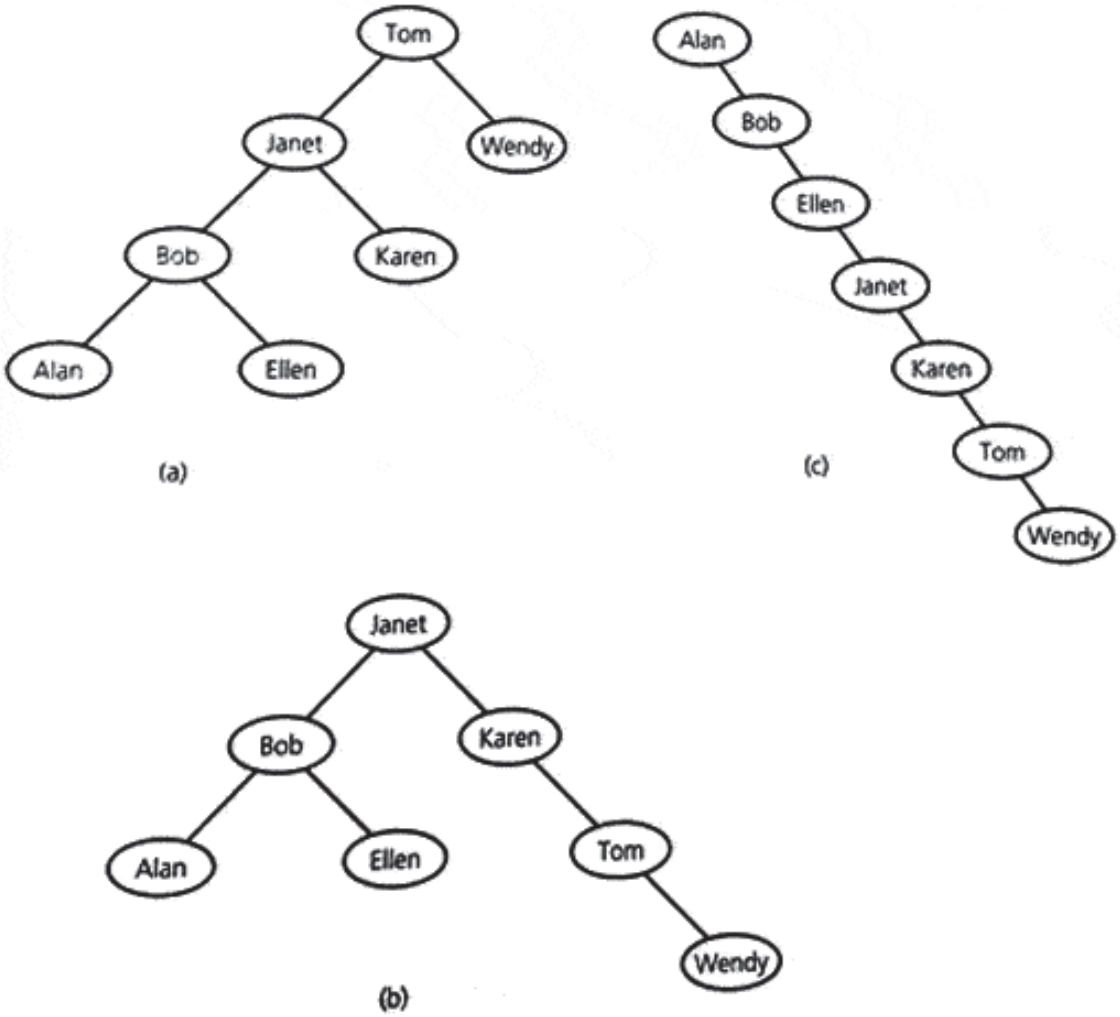


FIGURE 11-20
Binary search trees with the same data as in Figure 11-19

these trees have different shapes, the shape of the tree in no way affects the validity of the *search* algorithm. The algorithm requires only that a tree be a binary search tree.

The method *search* works more efficiently on some trees than on others, however. For example, with the tree in Figure 11-20c, the *search* algorithm inspects every node before locating Wendy. In fact, this binary tree really has the same structure as a sorted linear linked list and offers no advantage in efficiency. In contrast, with the full tree in Figure 11-19, the *search* algorithm inspects only the nodes that contain the names Janet, Tom, and Wendy. These names are exactly the names that a binary search of the sorted array in Figure 11-21 would inspect. Later in this chapter, you will learn more about how the shape of a binary search tree affects *search*'s efficiency and how the insertion and deletion operations affect this shape.

The algorithms that follow for insertion, deletion, retrieval, and traversal assume the reference-based implementation of a binary tree that was discussed earlier in this chapter. With minor changes, the basic algorithms also apply to other implementations of the binary tree. Also keep in mind the assumption that the items in the tree have unique search keys.

Insertion. Suppose that you want to insert a record for Frank into the binary search tree of Figure 11-19. As a first step, imagine that you instead want to *search* for the item with a search key of Frank. The *search* algorithm first searches the tree rooted at Janet, then the tree rooted at Bob, and then the tree rooted at Ellen. It then searches the tree rooted at the right child of Ellen. Because this tree is empty, as Figure 11-22 illustrates, the *search* algorithm has reached a base case and will terminate with the report that Frank is not present. What does it mean that *search* looked for Frank in the right subtree of Ellen? For one thing, it means that if Frank were the right child of Ellen, *search* would have found Frank there.

This observation indicates that a good place to insert Frank is as the right child of Ellen. Because Ellen has no right child, the insertion is simple, requiring only that Ellen's `rightChild` field reference a node containing Frank. More important, Frank belongs in this location—*search* will look for Frank here. Specifically, inserting Frank as the right child of Ellen will preserve the tree's binary search tree property. Because *search*, when searching for Frank, would follow a path that leads to the right child of Ellen, you are assured that Frank is in the proper relation to the names above it in the tree.

Use **search** to determine the insertion point

Using *search* to determine where in the tree to insert a new name always leads to an easy insertion. No matter what new item you insert into the tree, *search* will always terminate at an empty subtree. Thus, *search* always tells

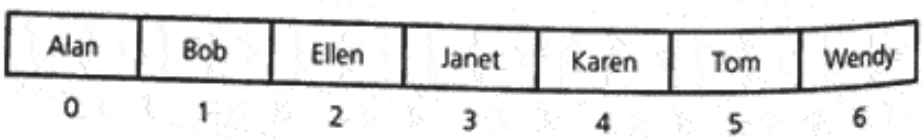


FIGURE 11-21
An array of names in sorted order

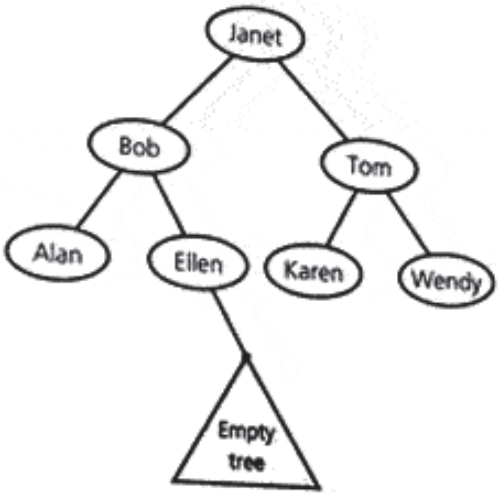


FIGURE 11-22
Empty subtree where ***search*** terminates

you to insert the item as a new leaf. Because adding a leaf requires only a change of the appropriate reference in the parent, the work required for an insertion is virtually the same as that for the corresponding search.

The following high-level pseudocode describes this insertion process:

First draft of the insertion algorithm

```
insertItem(in treeNode:TreeNode, in newItem:TreeItemType)
// Inserts newItem into the binary search tree of
// which treeNode is the root.
  Let parentNode be the parent of the empty subtree
      at which search terminates when it seeks
      newItem's search key

  if (Search terminated at parentNode's left subtree) {
    Set leftChild of parentNode to reference newItem
  }
  else {
    Set rightChild of parentNode to reference newItem
  }  // end if
```

The appropriate reference—*leftChild* or *rightChild*—of node *parentNode* must be set to reference the new node. The recursive nature of the *search* algorithm provides an elegant means of setting the reference, provided that you return *treeNode* as the result of the method, as you will see. Thus, *insertItem* is refined as follows:

Refinement of the insertion algorithm

```
+insertItem(in treeNode:TreeNode, in newItem:TreeItemType)
// Inserts newItem into the binary search tree of
// which treeNode is the root.
```

```
if (treeNode is null) {
  Create a new node and let treeNode reference it
  Create a new node with newItem as the data portion
  Set the references in the new node to null
}

else if (newItem.getKey() < treeNode.item.getKey()) {
  treeNode.leftChild = insertItem(treeNode.leftChild, newItem)
}
else {
  treeNode.rightChild = insertItem(treeNode.rightChild, newItem
}     end if

return treeNode
```

How does this recursive algorithm set *leftChild* and *rightChild* to reference the new node? The situation is quite similar to the recursive insertion method for the sorted linked list that you saw in Chapter 5. If the tree was empty before the insertion, the external reference to the root of the tree would be *null* and the method would not make a recursive call. When this situation occurs, a new tree node must be created and initialized. The method then returns the reference to this node, which should be made the new root of the tree. Figure 11-23a illustrates insertion into an empty tree.

The general case of *insertItem* is similar to the special case for an empty tree. When the formal parameter *treeNode* becomes *null*, the corresponding actual argument is the *leftChild* or *rightChild* reference in the parent of the empty subtree; that is, this reference has the value *null*. This reference was passed to the *insertItem* method by one of the recursive calls

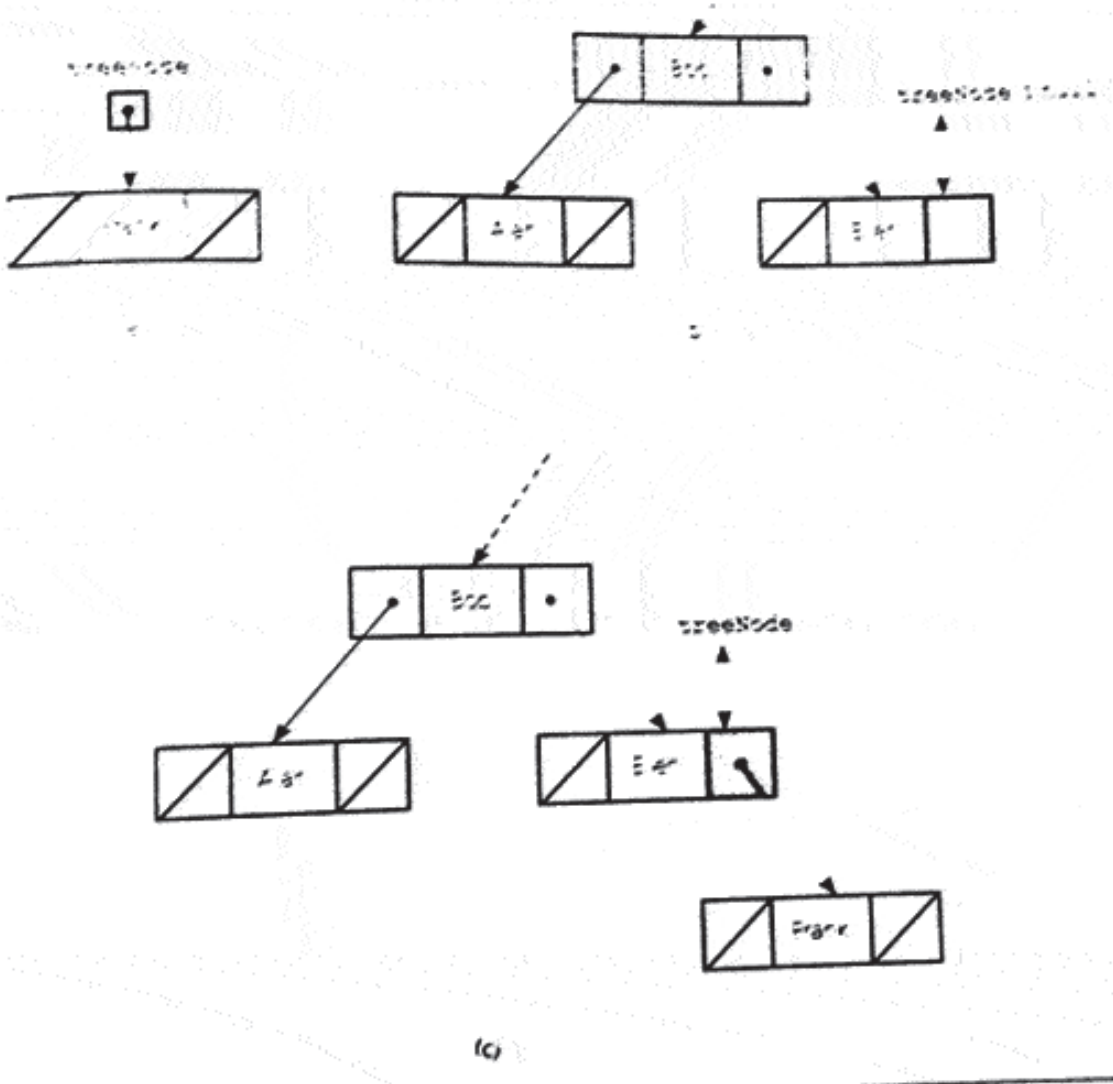
```
insertItem(treeNode.leftChild, newItem)
```

or

```
insertItem(treeNode.rightChild, newItem)
```

As in the case of the empty tree, the method will return the reference to a new node, which then must be set as either the parent's left child or the parent's right child. Thus, you complete the insertion by using either *leftChild* to make the node the new left child of the parent, or *rightChild* to make the node the new right child of the parent. Parts *b* and *c* of Figure 11-23 illustrate the general case of insertion.

You can use *insertItem* to create a binary search tree. For example, beginning with an empty tree, if you insert the names Janet, Bob, Alan, Ellen, Tom, Karen, and Wendy in order, you will get the binary search tree in Figure 11-19. It is interesting to note that the names Janet, Bob, Alan, Ellen, Tom, Karen, and Wendy constitute the preorder traversal of the tree in Figure 11-19.



(c)

FIGURE 11-23
(a) copy of one an empty tree; (b) search terminates at a leaf; (c) insertion at a leaf

To copy a tree, traverse it in preorder and insert each item visited into a new tree

Thus, if you take the output of a preorder traversal of a binary search tree and use it with *insertItem* to create a binary search tree, you will obtain a duplicate tree.

By inserting the previous names in a different order, you will get a different binary search tree. For example, by inserting the previous names in alphabetic order, you will get the binary search tree in Figure 11-20c.

Deletion. The deletion operation is a bit more involved than insertion. First, you use the *search* algorithm to locate the item with the specified search key

First draft of the deletion algorithm

and then, if it is found, you must remove the item from the tree. A first draft of the algorithm follows:

```
+deleteItem(in rootNode:TreeNode, in searchKey:KeyType)
// Deletes from the binary search tree (with root
// rootNode) the item whose search key equals
// searchKey. If no such item exists, the operation
// fails and throws TreeException.
  Locate (by using the search algorithm) the item i
      whose search key equals searchKey

  if (item i is found) {
    Remove item i from the tree
  }
  else {
    throw a tree exception
  }  // end if
```

The essential task here is

```
Remove item i from the tree
```

Three cases for the node *N* containing the item to be deleted

Assuming that `deleteItem` locates item *i* in a particular node *N*, there are three cases to consider:

1. *N* is a leaf.
2. *N* has only one child.
3. *N* has two children.

Case 1: Set the reference in a leaf's parent to `null`

The first case is the easiest. To remove the leaf containing item *i*, you need only set the reference in its parent to `null`. The second case is a bit more difficult. If *N* has only one child, you have two possibilities:

Case 2: Two possibilities for a node with one child

- *N* has only a left child.
- *N* has only a right child.

The two possibilities are symmetrical, so it is sufficient to illustrate the solution for a left child. In Figure 11-24a, *L* is the left child of *N*, and *P* is the parent of *N*. *N* can be either the left child or the right child of *P*. If you deleted *N* from the tree, *L* would be without a parent, and *P* would be without one of its children. Suppose you let *L* take the place of *N* as one of *P*'s children, as in Figure 11-24b. Does this adoption preserve the binary search tree property?

Let *N*'s parent adopt *N*'s child

If *N* is the left child of *P*, for example, all of the search keys in the subtree rooted at *N* are less than the search key in *P*. Thus, all of the search keys in the subtree rooted at *L* are less than the search key in *P*. Therefore, after *N* is removed and *L* is adopted by *P*, all of the search keys in *P*'s left subtree are still less than the search key in *P*. This deletion strategy thus preserves the binary

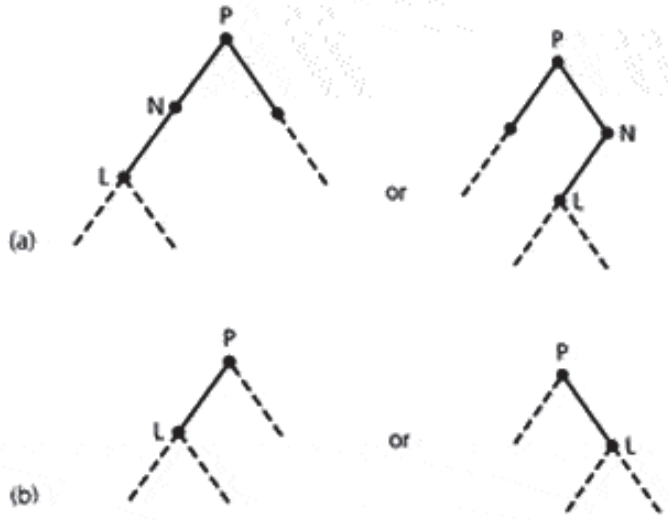


FIGURE 11-24

(a) *N* with only a left child—*N* can be either the left or right child of *P*; (b) after deleting node *N*

search tree property. A parallel argument holds if *N* is a right child of *P*, and therefore the binary search tree property is preserved in either case.

Case 3: *N* has two children

The most difficult of the three cases occurs when the item to be deleted is in a node *N* that has two children, as in Figure 11-25. As you just saw, when *N* has only one child, the child replaces *N*. However, when *N* has two children, these children cannot both replace *N*: *N*'s parent has room for only one of *N*'s children as a replacement for *N*. A different strategy is necessary.

In fact, you will not delete *N* at all. You can find another node that is easier to delete and delete it instead of *N*. This strategy may sound like cheating. After all, the programmer who requests

```
nameTree.delete(searchKey)
```

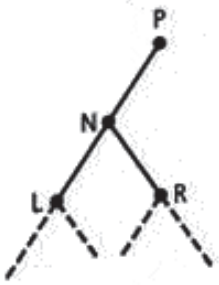


FIGURE 11-25

N with two children

expects that the item whose search key equals `searchKey` will be deleted from the ADT binary search tree. However, the programmer expects only that the *item* will be deleted and has no right, because of the wall between the program and the ADT implementation, to expect a particular *node* in the tree to be deleted.

Deleting an item whose node has two children

Consider, then, an alternate strategy. To delete from a binary search tree an item that resides in a node N that has two children, take the following steps:

1. Locate another node M that is easier to remove from the tree than the node N.
2. Copy the item that is in M to N, thus effectively deleting from the tree the item originally in N.
3. Remove the node M from the tree.

What kind of node M is easier to remove than the node N? Because you know how to delete a node that has no children or one child, M could be such a node. You have to be careful, though. Can you choose any node and copy its data into N? No, because you must preserve the tree's status as a binary search tree. For example, if in the tree of Figure 11-26, you copied the data from M to N, the result would no longer be a binary search tree.

What data item, when copied into the node N, will preserve the tree's status as a binary search tree? All of the search keys in the left subtree of N are less than the search key in N, and all of the search keys in the right subtree of N are greater than the search key in N. You must retain this property when you replace the search key x in node N with the search key y. There are two suitable possibilities for the value y. It can come immediately after or immediately before x in the sorted order of search keys. If y comes immediately after x, then clearly all search keys in the left subtree of N are smaller than y, because they are all smaller than x, as Figure 11-27 illustrates. Further, all search keys

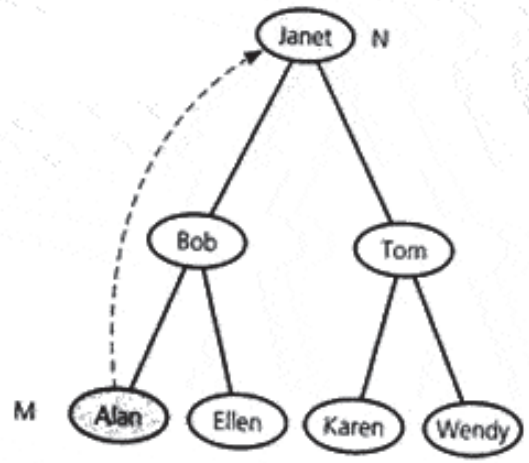


FIGURE 11-26
Not any node will do

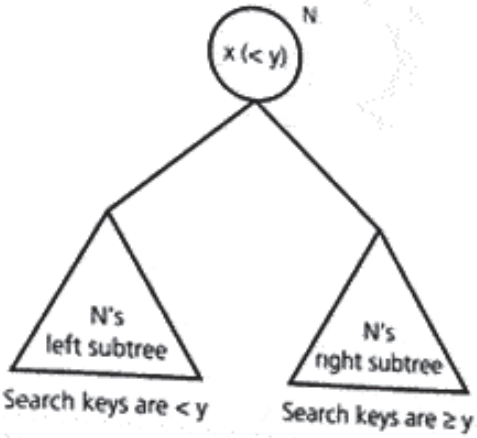


FIGURE 11-27
Search key x can be replaced by y

in the right subtree of N are greater than or equal to y, because they are greater than x and, by assumption, there are no search keys in the tree between x and y. A similar argument illustrates that if y comes immediately before x in the sorted order, it is greater than or equal to all search keys in the left subtree of N and smaller than all search keys in the right subtree of N.

You can thus copy into N either the item whose search key is immediately after N's search key[4] or the item whose search key is immediately before it. Suppose that, arbitrarily, you decide to use the node whose search key y comes immediately after N's search key x. This search key is called x's **inorder successor.**[5] How can you locate this node? Because N has two children, the inorder successor of its search key is in the leftmost node of N's right subtree. That is, to find the node that contains y, you follow N's `rightChild` reference to its right child R, which must be present because N has two children. You then descend the tree rooted at R by taking left branches at each node until you encounter a node M with no left child. You copy the item in this node M into node N and then, because M has no left child, you can remove M from the tree as one of the two easy cases. (See Figure 11-28.)

The inorder successor of N's search key is in the leftmost node in N's right subtree

A more detailed high-level description of the deletion algorithm follows:

Second draft of the deletion algorithm

```
+deleteItem(in rootNode:TreeNode, in searchKey:KeyType)
// Deletes from the binary search tree (with root rootNode)
// the item whose search key equals searchKey. If no such
// item exists, the operation fails and throws TreeException.

  Locate (by using the search algorithm) the
     item whose search key equals searchKey; it
     occurs in node i
```

4. N's search key is the search key of the data item in N.
5. We also will use the term N's inorder successor to mean the inorder successor of N's search key.

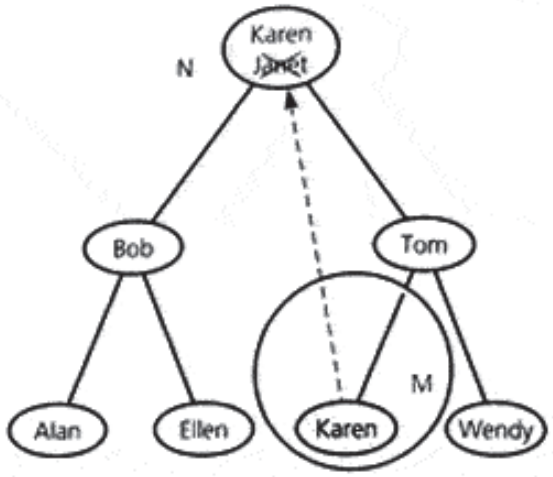


FIGURE 11-28
Copying the item whose search key is the inorder successor of *N*'s search key

```
  if (item is found in node i) {
    deleteNode(i)  // defined next
  }
  else {
    throw a tree exception
  }  // end if

+deleteNode(in treeNode:TreeNode):TreeNode
// Deletes the item in treeNode. Returns
// the root node of the resulting tree.

  if (treeNode is a leaf) {
    Remove treeNode from the tree
  }
  else if (treeNode has only one child c) {
    if (c was a left child of its parent p){
      Make c the left child of p
    }
    else {
      Make c the right child of p
    }  // end if
  }
  else { // treeNode has two children
    Find the item contained in treeNode's
        inorder successor
    Copy the item into treeNode
    Remove treeNode's inorder successor by using the
        previous technique for a leaf or a node
        with one child
  }  // end if
  return reference to root node of resulting tree
```

In the following refinement, `search`'s algorithm is adapted and inserted directly into `deleteItem`. Also, the method `deleteNode` uses the method `findLeftmost` to find the item in the node *M*, that is, the inorder successor of node *N*. The item in *M* is saved for later replacement of the item in node *N*. Next, `deleteLeftmost` deletes *M* from the tree. The saved item then replaces the item in node *N*, thus deleting it from the binary search tree.

Final draft of the deletion algorithm

```
+deleteItem(in rootNode:TreeNode, in searchKey:KeyType):TreeNode
  Deletes from the binary search tree (with root
  rootNode) the item whose search key equals searchKey.
  Returns the root node of the resulting tree.
  If no such item exists, the operation
  fails and throws TreeException.

  if (rootNode is null) {
    throw TreeException // item not found
  }
  else if (searchKey equals the key in rootNode item) {
     delete the rootNode; a new root of the tree is
     returned
    newRoot = deleteNode(rootNode, searchKey)
    return newRoot
  }
  else if (searchKey is less than the key in rootNode item) {
    newLeft = deleteItem(rootNode.leftChild, searchKey)
    rootNode.leftChild = newLeft
    return rootNode // returns rootNode with new left
                    // subtree
  }
  else { // search the right subtree
    newRight = deleteItem(rootNode.rightChild, searchKey)
    rootNode.rightChild = newRight
    return rootNode // returns rootNode with new right
                    // subtree
} // end if

+deleteNode(in treeNode:TreeNode):TreeNode
// Deletes the item in the node referenced by treeNode.
// Returns the root node of the resulting tree.

  if (treeNode is a leaf) {
    // remove leaf from the tree
   return null
  }
```

```
    else if (treeNode has only one child c) {
      // c replaces treeNode as the child of
      // treeNode's parent
      if (c is the left child of treeNode) {
        return treeNode.leftChild
      }
      else {
        return treeNode.rightChild
      } // end if
    }
    else { // treeNode has two children
      // find the inorder successor of the search key in
      // treeNode: it is in the leftmost node of the
      // subtree rooted at treeNode's right child
      replacementItem = findLeftMost(treeNode.rightChild)
      replacementRChild = deleteLeftmost(treeNode.rightChild)
      Set treeNode's item to replacementItem
      Set treeNode's right child to replacementRChild
      return treeNode
    } // end if

+findLeftmost(in treeNode:TreeNode):TreeNode
// Returns the item that is the leftmost
// descendant of the tree rooted at treeNode.
  if (treeNode.leftChild == null) {
    // this is the node you want, so return its item
    return treeNode.item
  }
  else {
    return findLeftmost(treeNode.leftChild)
  } // end if

+deleteLeftmost(in treeNode:TreeNode):TreeNode
// Deletes the node that is the leftmost
// descendant of the tree rooted at treeNode.
// Returns subtree of deleted node.

  if (treeNode.leftChild == null) {
    // this is the node you want; it has no left
    // child, but it might have a right subtree

    // the return value of this method is a
    // child reference of treeNode's parent; thus, the
    // following "moves up" treeNode's right subtree
    return treeNode.rightChild
  }
```

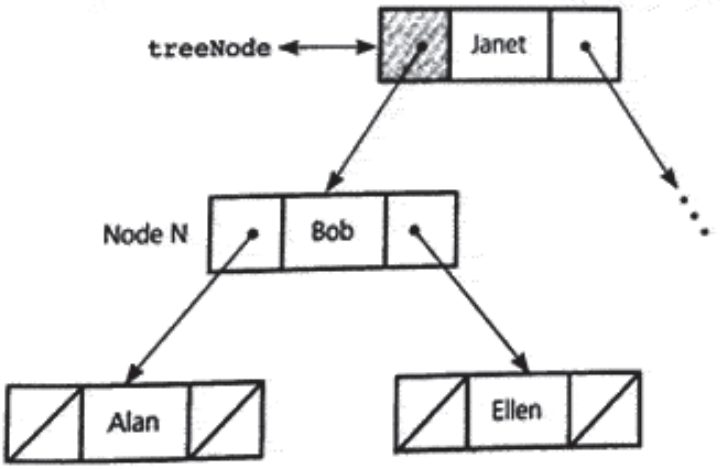
```
else {
  replacementLChild = deleteLeftmost(treeNode.leftChild)
  treeNode.leftChild = replacementLChild
  return treeNode
} // end if
```

Observe that, as in the case of the `insertItem` method, the actual argument that corresponds to `rootNode` either is one of the references of the parent of X, as Figure 11-29 depicts, or is the external reference to the root, in the case where X is the root of the original tree. In either case `rootNode` references X. Thus, any change you make to `rootNode` by calling the method `deleteNode` with actual argument `rootNode` changes the reference in the parent of X. The recursive method `deleteLeftmost`, which is called by `deleteNode` if X has two children, also uses this strategy to remove the inorder successor of the node containing the item to be deleted.

Exercise 30 at the end of this chapter describes an easier deletion algorithm. However, that algorithm tends to increase the height of the tree, and, as you will see later, an increase in height can decrease the efficiency of searching the tree.

Retrieval. By refining the `search` algorithm, you can implement the retrieval operation. Recall that the `search` algorithm is

```
+search(in bst:BinarySearchTree, in searchKey:KeyType)
  Searches the binary search tree bst for the item
  whose search key is searchKey.

 if (bst is empty) {
   The desired record is not found
 }
```



Any change to `treeNode` while deleting node N (Bob) changes `leftChild` of Janet

FIGURE 11-29
Recursive deletion of node *N*

```
  else if (searchKey == search key of bst's root item) {
    The desired record is found
  }
  else if (searchKey < search key of bst's root item) {
    search(Left subtree of bst, searchKey)
  }
  else {
    search(Right subtree of bst, searchKey)
  }  // end if
```

retrieveItem is a refinement of **search**

The retrieval operation must return the item with the desired search key if it exists; otherwise it must return a *null* reference. The retrieval algorithm, therefore, appears as follows:

```
+retrieveItem(in treeNode:TreeNode,
              in searchKey:KeyType):TreeItemType
// Returns the item (treeItem) whose search
// key equals searchKey from the binary search tree
// that has treeNode as its root. The operation
// returns a null reference if no such item
// exists.

 if (treeNode == null) {
   treeItem = null  // tree is empty
 }
 else if (searchKey == treeNode.item.getKey()) {
   // item is in the root of some subtree
   treeItem = treeNode.item
 }
 else if (searchKey < treeNode.item.getKey()) {
   // search the left subtree
   treeItem =
   retrieveItem(treeNode.leftChild, searchKey)
 }
 else { // search the right subtree
   treeItem =
   retrieveItem(treeNode.rightChild, searchKey)
 }  // end if

 return treeItem
```

Traversal. The traversals for a binary search tree are the same as the traversals for a binary tree. You should be aware, however, that an inorder traversal of

a binary search tree will visit the tree's nodes in sorted search-key order. Before seeing the proof of this statement, recall the inorder traversal algorithm:

```
+inorder(in bst:BinarySearchTree)
// Traverses the binary tree bst in inorder.

  if (bst is not empty) {
    inorder(Left subtree of bst's root)
    Process the root of bst
    inorder(Right subtree of bst's root)
  } // end if
```

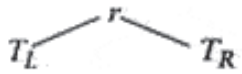
THEOREM 11-1. The inorder traversal of a binary search tree T will visit its nodes in sorted search-key order.

PROOF. The proof is by induction on h, the height of T.

Basis: $h = 0$. When T is empty, the algorithm does not visit any nodes. This is the proper sorted order for the zero names that are in the tree!

Inductive hypothesis: Assume that the theorem is true for all k, $0 < k < h$. That is, assume for all k ($0 < k < h$) that the inorder traversal visits the nodes in sorted search-key order.

Inductive conclusion: You must show that the theorem is true for $k = h > 0$. T has the form



Because T is a binary search tree, all the search keys in the left subtree T_L are less than the search key in the root r, and all the search keys in the right subtree T_R are greater than the search key in r. The `inorder` algorithm will visit all the nodes in T_L, then visit r, and then visit all the nodes in T_R. Thus, the only concern is that `inorder` visit the nodes within each of the subtrees T_L and T_R in the correct sorted order. But because T is a binary search tree of height h, each subtree is a binary search tree of height less than h. Therefore, by the inductive hypothesis, `inorder` visits the nodes in each subtree T_L and T_R in the correct sorted search-key order. **(End of proof.)**

It follows from this theorem that `inorder` visits a node's inorder successor immediately after it visits the node.

Use inorder traversal to visit nodes of a binary search tree in search-key order

A Reference-Based Implementation of the ADT Binary Search Tree

A Java reference-based implementation of the ADT binary search tree follows. Notice the protected methods that implement the recursive algorithms. These methods are not public, because clients do not have access to node references.

The methods could be private, but making them protected enables a derived class to use them directly.

```
import SearchKeys.KeyedItem;

// ADT binary search tree.
//  Assumption: A tree contains at most one item with a
//              given search key at any time.

public class BinarySearchTree<T extends KeyedItem<KT>,
                              KT extends Comparable<? super KT>>
       extends BinaryTreeBasis<T> {
  // inherits isEmpty(), makeEmpty(), getRootItem(), and
  // the use of the constructors from BinaryTreeBasis

  public BinarySearchTree() {
  }  // end default constructor

  public BinarySearchTree(T rootItem) {
    super(rootItem);
  }  // end constructor

  public void setRootItem(T newItem)
              throws UnsupportedOperationException {
    throw new UnsupportedOperationException();
  }  // end setRootItem

  public void insert(T newItem) {
    root = insertItem(root, newItem);
  }  // end insert

  public T retrieve(KT searchKey) {
    return retrieveItem(root, searchKey);
  }  // end retrieve

  public void delete(KT searchKey) throws TreeException {
    root = deleteItem(root, searchKey);
  }  // end delete

  public void delete(T item) throws TreeException {
    root = deleteItem(root, item.getKey());
  }  // end delete

  protected TreeNode<T> insertItem(TreeNode<T> tNode,
                                   T newItem) {
    TreeNode<T> newSubtree;
    if (tNode == null) {
```

```
      // position of insertion found; insert after leaf
      // create a new node
      tNode = new TreeNode<T>(newItem, null, null);
      return tNode;
    }  // end if
    T nodeItem = tNode.item;

    // search for the insertion position

    if (newItem.getKey().compareTo(nodeItem.getKey()) < 0) {
      // search the left subtree
      newSubtree = insertItem(tNode.leftChild, newItem);
      tNode.leftChild = newSubtree;
      return tNode;
    }
    else {  // search the right subtree
      newSubtree = insertItem(tNode.rightChild, newItem);
      tNode.rightChild = newSubtree;
      return tNode;
    }  // end if
  }  // end insertItem

  protected T retrieveItem(TreeNode<T> tNode,
                           KT searchKey) {
    T treeItem;
    if (tNode == null) {
      treeItem = null;
    }
    else {
      T nodeItem = tNode.item;
      if (searchKey.compareTo(nodeItem.getKey()) == 0) {
        // item is in the root of some subtree
        treeItem = tNode.item;
      }
      else if (searchKey.compareTo(nodeItem.getKey()) < 0) {
        // search the left subtree
        treeItem = retrieveItem(tNode.leftChild, searchKey);
      }
      else { // search the right subtree
        treeItem = retrieveItem(tNode.rightChild, searchKey);
      }  // end if
    }  // end if
    return treeItem;
  }  // end retrieveItem

  protected TreeNode<T> deleteItem(TreeNode<T> tNode,
                                   KT searchKey) {

    // Calls: deleteNode.
```

```
    TreeNode<T> newSubtree;
    if (tNode == null) {
      throw new TreeException("TreeException: Item not found");
    }
    else {
      T nodeItem = tNode.item;
      if (searchKey.compareTo(nodeItem.getKey()) == 0) {
        // item is in the root of some subtree
        tNode = deleteNode(tNode);  // delete the item
      }
      // else search for the item
      else if (searchKey.compareTo(nodeItem.getKey()) < 0) {
        // search the left subtree
        newSubtree = deleteItem(tNode.leftChild, searchKey);
        tNode.leftChild = newSubtree;
      }
      else { // search the right subtree
        newSubtree = deleteItem(tNode.rightChild, searchKey);
        tNode.rightChild = newSubtree;
      }  // end if
    }  // end if
    return tNode;
  }  // end deleteItem

  protected TreeNode<T> deleteNode(TreeNode<T> tNode) {
    // Algorithm note: There are four cases to consider:
    //   1. The tNode is a leaf.
    //   2. The tNode has no left child.
    //   3. The tNode has no right child.
    //   4. The tNode has two children.
    // Calls: findLeftmost and deleteLeftmost
    T replacementItem;

    // test for a leaf
    if ( (tNode.leftChild == null) &&
         (tNode.rightChild == null) ) {
      return null;
    }  // end if leaf

    // test for no left child
    else if (tNode.leftChild == null) {
      return tNode.rightChild;
    }  // end if no left child

    // test for no right child
    else if (tNode.rightChild == null) {
```

```
      return tNode.leftChild;
    }  // end if no right child

       there are two children:
       retrieve and delete the inorder successor
    else {
      replacementItem = findLeftmost(tNode.rightChild);
      tNode.item = replacementItem;
      tNode.rightChild = deleteLeftmost(tNode.rightChild);
      return tNode;
    }  // end if
  }  // end deleteNode

  protected T findLeftmost(TreeNode<T> tNode)  {
    if (tNode.leftChild == null) {
      return tNode.item;
    }
    else {
      return findLeftmost(tNode.leftChild);
    }  // end if
  }  // end findLeftmost

  protected TreeNode<T> deleteLeftmost(TreeNode<T> tNode) {
    if (tNode.leftChild == null) {
      return tNode.rightChild;
    }
    else {
      tNode.leftChild = deleteLeftmost(tNode.leftChild);
      return tNode;
    }  // end if
  }  // end deleteLeftmost

}  // end BinarySearchTree
```

The class *TreeIterator* developed earlier in the chapter can be used with *BinarySearchTree*. It would also make sense to implement the *remove* method, since *BinarySearchTree* provides a method for deleting a node from the tree. Exercise 32 at the end of this chapter asks you to explore this possibility.

The Efficiency of Binary Search Tree Operations

You have seen binary search trees in many shapes. For example, even though the binary search trees in Figures 11-19 and 11-20c have seven nodes each, they have radically different shapes and heights. You saw that to locate Wendy in Figure 11-20c, you would have to inspect all seven nodes, but you can locate Wendy in Figure 11-19 by inspecting only three nodes (Jane, [illegible], and Wendy). Consider now the relationship between the height of a binary search tree and the efficiency of the retrieval, insertion, and deletion operations

Each of these operations compares the specified value `searchKey` to the search keys in the nodes along a **path** through the tree. This path always starts at the root of the tree and, at each node N, follows the left or right branch, depending on the comparison of `searchKey` to the search key in N. The path terminates at the node that contains `searchKey` or, if `searchKey` is not present, at an empty subtree. Thus, each retrieval, insertion, or deletion operation requires a number of comparisons equal to the number of nodes along this path. This means that the maximum number of comparisons that each operation can require is the number of nodes on the longest path through the tree. In other words, the *maximum number of comparisons that these operations can require is equal to the height of the binary search tree.* What, then, are the maximum and minimum heights of a binary search tree of n nodes?

The maximum number of comparisons for a retrieval, insertion, or deletion is the height of the tree

The maximum and minimum heights of a binary search tree. You can maximize the height of a binary tree with n nodes simply by giving each internal node (nonleaf) exactly one child, as shown in Figure 11-30. This process will result in a tree of height n. An n-node tree with height n strikingly resembles a linear linked list.

n is the maximum height of a binary tree with n nodes

A minimum-height binary tree with n nodes is a bit more difficult to obtain. As a first step, consider the number of nodes that a binary tree with a given height h can have. For example, if $h = 3$, the possible binary trees include those in Figure 11-31. Thus, binary trees of height 3 can have between 3 and 7 nodes. In addition, Figure 11-31 shows that 3 is the minimum height for a binary tree with 4, 5, 6, or 7 nodes. Similarly, binary trees with more than 7 nodes require a height greater than 3.

Except for the last level, each level of a minimum-height binary tree must contain as many nodes as possible

Intuitively, to minimize the height of a binary tree given n nodes, you must fill each level of the tree as completely as possible. A complete tree meets this requirement (although it does not matter whether the nodes on the last level are filled left

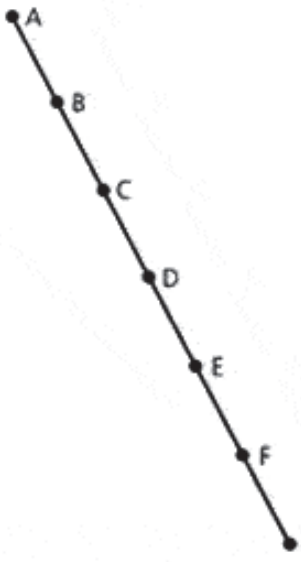


FIGURE 11-30
A maximum-height binary tree with seven nodes

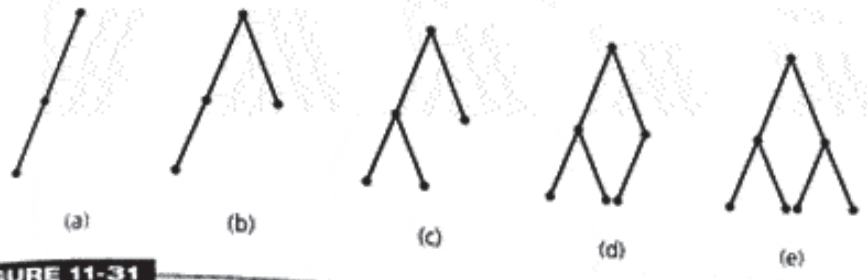


FIGURE 11-31
Binary trees of height 3

to right). In fact, trees *b*, *c*, *d*, and *e* of Figure 11-31 are complete trees. If a complete binary tree of a given height *h* is to have the maximum possible number of nodes, it should be full (as in Figure 11-31e). Figure 11-32 counts these nodes by level and shows the following:

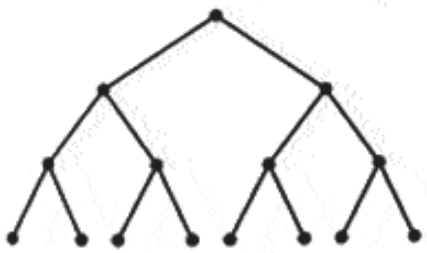
THEOREM 11-2. A full binary tree of height $h \geq 0$ has $2^h - 1$ nodes.

A formal proof by induction of this theorem is left as an exercise.

It follows then that

THEOREM 11-3. The maximum number of nodes that a binary tree of height *h* can have is $2^h - 1$.

You cannot add nodes to a full binary tree of height *h* without increasing its height. The formal proof of this theorem, which closely parallels that of Theorem 11-2, is left as an exercise.



Level	Number of nodes at this level	Number of nodes at this and previous levels
1	$1 = 2^0$	$1 = 2^1 - 1$
2	$2 = 2^1$	$3 = 2^2 - 1$
3	$4 = 2^2$	$7 = 2^3 - 1$
4	$8 = 2^3$	$15 = 2^4 - 1$
.	.	.
.	.	.
.	.	.
h	2^{h-1}	$2^h - 1$

FIGURE 11-32
Counting the nodes in a full binary tree of height *h*

The following theorem uses Theorems 11-2 and 11-3 to determine the minimum height of a binary tree that contains some given number of nodes.

THEOREM 11-4. The minimum height of a binary tree with n nodes is $\lceil \log_2(n+1) \rceil$.[6]

PROOF. Let h be the smallest integer such that $n \leq 2^h - 1$. To find the minimum height of a binary tree with n nodes, first establish the following facts:

1. *A binary tree whose height is* $\leq h - 1$ *has* $< n$ *nodes.*

   By Theorem 11-3, a binary tree of height $h - 1$ has at most $2^{h-1} - 1$ nodes. If it is possible that $n \leq 2^{h-1} - 1 < 2^h - 1$, then h is not the smallest integer such that $n \leq 2^h - 1$. Therefore, n must be greater than $2^{h-1} - 1$ or, equivalently, $2^{h-1} - 1 < n$. Because a binary tree of height $h - 1$ has at most $2^{h-1} - 1$ nodes, it must have fewer than n nodes.

2. *There exists a complete binary tree of height* h *that has exactly* n *nodes.*

   Consider the full binary tree of height $h - 1$. By Theorem 11-2, it has $2^{h-1} - 1$ nodes. As you just saw, $n > 2^{h-1} - 1$ because h was selected so that $n \leq 2^h - 1$. You can thus add nodes to the full tree from left to right until you have n nodes, as Figure 11-33 illustrates. Because $n \leq 2^h - 1$ and a binary tree of height h cannot have more than $2^h - 1$ nodes, you will reach n nodes by the time level h is filled up.

3. *The minimum height of a binary tree with* n *nodes is the smallest integer* h *such that* $n \leq 2^h - 1$.

   If h is the smallest integer such that $n \leq 2^h - 1$, and if a binary tree has height $\leq h - 1$, then by fact 1, it has fewer than n nodes. Because by fact 2 there is a binary tree of height h that has exactly n nodes, h must be as small as possible.

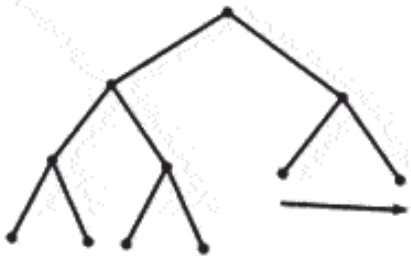


FIGURE 11-33
Filling in the last level of a tree

6. The ceiling of X, which $\lceil X \rceil$ denotes, is X rounded up. For example, $\lceil 6 \rceil = 6$, $\lceil 6.1 \rceil = 7$, and $\lceil 6.8 \rceil = 7$.

The previous discussion implies that

$$h - 1 < n \leq 2^h - 1$$
$$h - 1 < n + 1 \leq 2^h$$
$$h - 1 < \log_2(n + 1) \leq h$$

If $\log_2(n + 1) = h$, the theorem is proven. Otherwise, $h - 1 < \log_2(n + 1) < h$ implies that $\log_2(n + 1)$ cannot be an integer. Therefore, round $\log_2(n + 1)$ up to get h.

Thus, $h = \lceil \log_2(n + 1) \rceil$ is the minimum height of a binary tree with n nodes. **(End of proof.)**

Complete trees and full trees have minimum height

Complete trees and full trees with n nodes thus have heights of $\lceil \log_2(n + 1) \rceil$, which, as you just saw, is the theoretical minimum. This minimum height is the same as the maximum number of comparisons a binary search must make to search an array with n elements. Thus, if a binary search tree is complete and therefore balanced, the time it takes to search it for a value is about the same as that required for a binary search of an array. On the other hand, as you go from balanced trees toward trees with a linear structure, the height approaches the number of nodes n. This number is the same as the maximum number of comparisons that you must make when searching a linked list of n nodes.

The height of an n-node binary search tree ranges from $\lceil \log_2(n + 1) \rceil$ to n

However, the outstanding efficiency of the operations on a binary search tree hinges on the assumption that the height of the binary search tree is $\lceil \log_2(n + 1) \rceil$. What will the height of a binary search tree actually be? The factor that determines the height of a binary search tree is the order in which you perform the insertion and deletion operations on the tree. Recall that, starting with an empty tree, if you insert names in the order Alan, Bob, Ellen, Janet, Karen, Tom, Wendy, you would obtain a binary search tree of maximum height, as shown in Figure 11-20c. On the other hand, if you insert names in the order Janet, Bob, Tom, Alan, Ellen, Karen, Wendy, you would obtain a binary search tree of minimum height, as shown in Figure 11-19.

Insertion in search-key order produces a maximum-height binary search tree

Which of these situations should you expect to encounter in the course of a real application? It can be proven mathematically that if the insertion and deletion operations occur in a random order, the height of the binary search tree will be quite close to $\log_2 n$. Thus, in this sense, the previous analysis is not unduly optimistic. However, in a real-world application, is it realistic to expect the insertion and deletion operations to occur in random order? In many applications, the answer is yes. There are, however, applications in which this assumption would be dubious. For example, the person preparing the previous sequence of names for the insertion operations might well decide to "help you out" by arranging the names to be inserted into sorted order. This arrangement, as has been mentioned, would lead to a tree of maximum height. Thus, while in many applications you can expect the behavior of a binary search tree to be excellent, you should be wary of the possibility of poor performance due to some characteristic of a given application.

Insertion in random order produces a near-minimum-height binary search tree

Is there anything you can do if you suspect that the operations might not occur in a random order? Similarly, is there anything you can do if you have an

enormous number of items and need to ensure that the height of the tree is close to $\log_2 n$? Chapter 13 presents variations of the basic binary search tree that are guaranteed always to remain balanced.

Figure 11-34 summarizes the order of the retrieval, insertion, deletion, and traversal operations for the ADT binary search tree.

Treesort

You can use the ADT binary search tree to sort an array of records efficiently into search-key order. To simplify the discussion, however, we will sort an array of integers into ascending order, as we did with the sorting algorithms in Chapter 10.

The basic idea of the algorithm is simple:

Treesort uses a binary search tree

```
treesort(inout anArray:ArrayType, in n:integer)
// Sorts the n integers in array anArray into
// ascending order.

  Insert anArray's elements into a binary search tree
    bTree

  Traverse bTree inorder. As you visit bTree's nodes, copy
    their data items into successive locations of anArray
```

An inorder traversal of the binary search tree `bTree` visits the integers in `bTree`'s nodes in ascending order.

A treesort can be quite efficient. As Figure 11-34 indicates, each insertion into a binary search tree requires $O(\log n)$ operations in the average case and $O(n)$ operations in the worst case. Thus, `treesort`'s n insertions require $O(n * \log n)$ operations in the average case and $O(n^2)$ operations in the worst case. The traversal of the tree involves one copy operation for each of the n elements and so is $O(n)$. Since $O(n)$ is less than $O(n * \log n)$ and $O(n^2)$, treesort is $O(n * \log n)$ in the average case and $O(n^2)$ in the worst case.

Treesort. Average case: $O(n * \log n)$; worst case: $O(n^2)$

Operation	Average case	Worst case
Retrieval	O(log n)	O(n)
Insertion	O(log n)	O(n)
Deletion	O(log n)	O(n)
Traversal	O(n)	O(n)

FIGURE 11-34

The order of the retrieval, insertion, deletion, and traversal operations for the reference-based implementation of the ADT binary search tree

Saving a Binary Search Tree in a File

Imagine a program that maintains the names, addresses, and telephone numbers of your friends and relatives. While the program is running, you can enter a name and get the person's address and phone number. If you terminate program execution, the program must save its database of people in a form that it can recover at a later time.

If the program uses a binary search tree to represent the database, it must save the tree's data in a file so that it can later restore the tree. You could save the tree by simply adding the `java.io.Serializable` interface to the various classes in the implementation of the binary search tree. Exercise 36 at the end of this chapter asks you to decide which classes would need to have this interface specified. Suppose, however, that you want to define the interface `java.io.Serializable` only on the items you are storing in the tree. We will consider two different algorithms for saving and restoring a binary search tree. The first algorithm restores a binary search tree to its original shape. The second restores a binary search tree to a shape that is balanced.

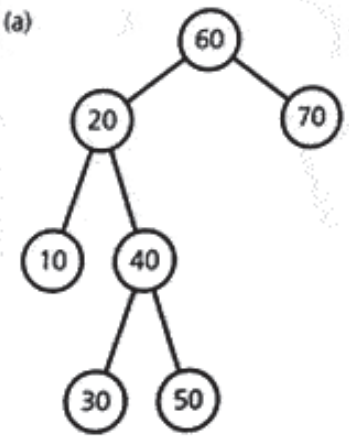
Saving a binary search tree and then restoring it to its original shape. The first algorithm restores a binary search tree to exactly the same shape it had before it was saved. For example, consider the tree in Figure 11-35a. If you save the tree in preorder, you get the sequence 60, 20, 10, 40, 30, 50, 70. If you then use the binary search tree `insert` to insert these values into a tree that is initially empty, you will get the original tree. Figure 11-35b shows this sequence of insertion operations in pseudocode.

Use a preorder traversal and **`insert`** to save and then restore a binary search tree in its original shape

Saving a binary search tree and then restoring it to a balanced shape. Can you do better than the previous algorithm? That is, do you necessarily want the restored tree to have its original shape? Recall that you can organize a given set of data items into binary search trees with many different shapes. Although the shape of a binary search tree has no effect whatsoever on the correctness of the ADT operations, it will affect the efficiency of those operations. Efficient operations are assured if the binary search tree is balanced.

A balanced binary search tree increases the efficiency of the ADT operations

(a)



(b)

```
bst.insert(60)
bst.insert(20)
bst.insert(10)
bst.insert(40)
bst.insert(30)
bst.insert(50)
bst.insert(70)
```

FIGURE 11-35
(a) A binary search tree **`bst`**; (b) the sequence of insertions that result in this tree

The algorithm that restores a binary search tree to a balanced shape is surprisingly simple. In fact, you can even guarantee a restored tree of minimum height—a condition stronger than balanced. To gain some insight into the solution, consider a full tree, because it is balanced. If you save a full tree in a file by using an inorder traversal, the file will be in sorted order, as Figure 11-36 illustrates. A full tree with exactly $n = 2^h - 1$ nodes for some height h has the exact middle of the data items in its root. The left and right subtrees of the root are full trees of $2^{h-1} - 1$ nodes each (that is, half of $n - 1$, since n is odd or, equivalently, $n/2$). Thus, you can use the following recursive algorithm to create a full binary search tree with n nodes, provided you either know or can determine n beforehand.

Building a full binary search tree

```
+readFull(in inputFile:FileType, in n:integer):TreeNode
// Builds a full binary search tree from n sorted values
// in a file and returns the tree's root.

  if (n > 0) {
    treeNode = a new node with null child references
    // construct the left subtree
    Set treeNode's left child to readFull(inputFile, n/2)

    // get the root
    Read item from file into treeNode's item

    // construct the right subtree
    Set treeNode's right child to readFull(inputFile, n/2)
  }  // end if

  return treeNode
```

Surprisingly, you can construct the tree directly by reading the sorted data sequentially from the file.

This algorithm for building a full binary search tree is simple, but what can you do if the tree to be restored is not full (that is, if it does not have $n = 2^h - 1$ nodes for some h)? The first thing that comes to mind is that the restored

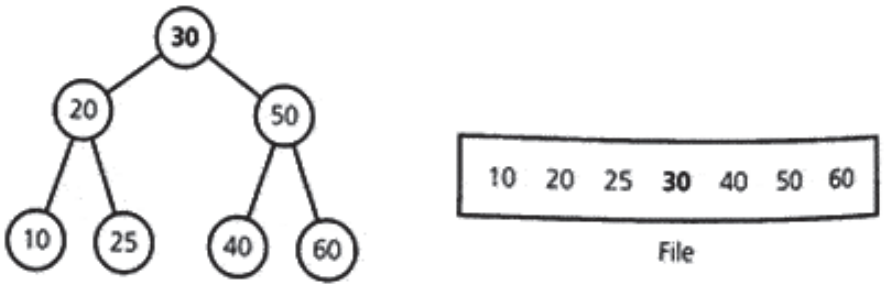


FIGURE 11-36
A full tree saved in a file by using inorder traversal

tree should be complete—full up to the last level, with the last level filled in from left to right. Actually, because you care only about minimizing the height of the restored tree, it does not matter where the nodes on the last level go, as Figure 11-37 shows.

The *readFull* algorithm is essentially correct even if the tree is not full. However, you do have to be a bit careful when computing the sizes of the left and right subtrees of the tree's root. If n is odd, both subtrees are of size $n/2$, as before. (The root is automatically accounted for.) If n is even, however, you have to account for the root and the fact that one of the root's subtrees will have one more node than the other. In this case, you can arbitrarily choose to put the extra node in the left subtree. The following algorithm makes these compensations:

Building a minimum-height binary search tree

```
+readTree(in inputFile:FileType, in n:integer):TreeNode
// Builds a minimum-height binary search tree from n sorted
// values in a file. Will return the tree's root.

 if (n > 0) {
   treeNode = reference to new node with null
              child references
   // construct the left subtree
   Set treeNode's left child to readTree(inputFile, n/2)

   // get the root
   Read item from file into treeNode's item

   // construct the right subtree
   Set treeNode's right child to
       readTree(inputFile, (n-1)/2)
 }  // end if

 return treeNode
```

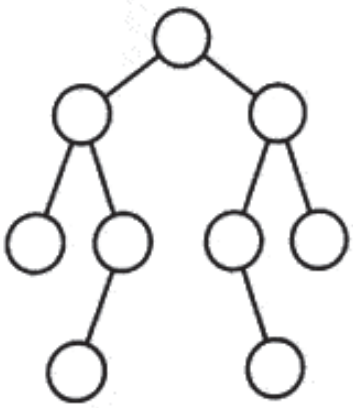


FIGURE 11-37
A tree of minimum height that is not complete

You should trace this algorithm and convince yourself that it is correct for both even and odd values of *n*.

To summarize, you can easily restore a tree as a balanced binary search tree if the data is sorted—that is, if it has been produced from the inorder traversal—and you know the number *n* of nodes in the tree. You need *n* so that you can determine the middle item and, in turn, the number of nodes in the left and right subtrees of the tree's root. Knowing these numbers is a simple matter of counting nodes as you traverse the tree and then saving the number in a file that the restore operation can read.

Note that `readTree` would be an appropriate protected method of `BinarySearchTree`, if you also had a public method to call it.

The JCF Binary Search Algorithm

The Java Collections Framework provides two binary search methods to find a specified element in a sorted `java.util.List`. The first is based upon the natural ordering of the elements:

```
static <T> int
  binarySearch(List<? extends Comparable<? super T>> list, T key)
```

The second is based upon a specified `Comparator`:

```
static <T> int
  binarySearch(List<? extends T> list, T key,
               Comparator<? super T> c)
```

The JCF `sort` methods shown in Chapter 10 can be used to sort the list before calling `binarySearch`. Both methods assume the list is in ascending order and if the element is found, its index in the list is returned (a value `>= 0`). If the element is not found, a negative value is returned. This value, `-(insertIndex)-1` can be used to determine the insertion point for the element in the sorted list, even if the element should be inserted at the end of the list. For example, if the `binarySearch` method returns –2, `insertIndex` would be 1.

The `binarySearch` methods run in logarithmic time if the elements in the `List` can be access directly in constant time (also known as random access). If the `List` does not implement the `RandomAccess` interface and is large, the `binarySearch` algorithm will do an iterator-based binary search that runs in linear time.

The following program demonstrates the use of the `binarySearch` algorithm on a sorted list:

```
import java.util.List;
import java.util.LinkedList;
import java.util.Collections;
import java.util.Arrays;

public class JCFSearchEx {
  public static void main(String args[]) {
```

```
    String[] names = {"Janet", "Michael", "Pat", "Craig",
                      "Andrew", "Sarah", "Evan", "Anita"};

    LinkedList<String> namelist = new LinkedList<String>();
    namelist.addAll(Arrays.asList(names));
    Collections.sort(namelist);
    String name = "Maite";

    int loc = Collections.binarySearch(namelist, name);
    if (loc < 0) {
      System.out.println(name + " should be inserted to position "
                                 + (-(loc+1)) + "\n");
      namelist.add(-(loc+1), name);
      System.out.println(namelist);
    } else {
      System.out.println(name + " was found in location "
                              + loc + "\n");
    }  // end if
  }  // end main
}  // end JCFSearchEx
```

11.4 General Trees

This chapter ends with a brief discussion of general trees and their relationship to binary trees. Consider the general tree in Figure 11-38. The three children *B*, *C*, and *D* of node *A*, for example, are siblings. The leftmost child *B* is called the **oldest child,** or **first child,** of *A*. One way to implement this tree uses the same node structure that we used for a reference-based binary tree. That is, each node has two references: The left one references the node's oldest child and the right one references the node's next sibling. Thus, you can use the data structure in Figure 11-39 to implement the tree in Figure 11-38. Notice that the structure in Figure 11-39 also represents the binary trees:general trees:general treestree pictured in Figure 11-40.

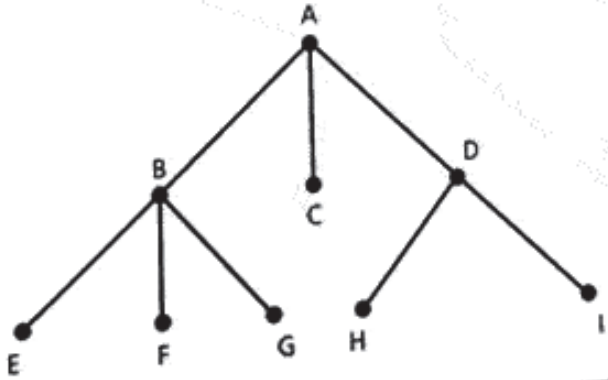


FIGURE 11-38
A general tree

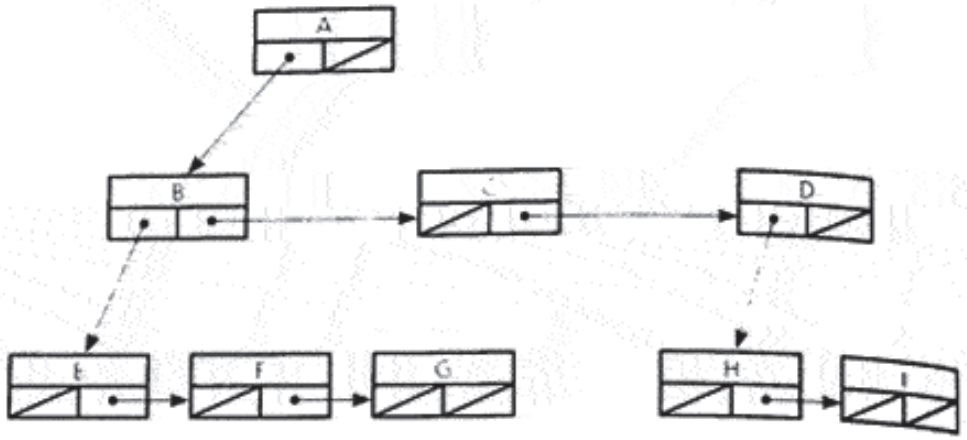


FIGURE 11-39
A reference-based implementation of the general tree in Figure 11-38

An ***n*-ary tree** is a generalization of a binary tree whose nodes each can have no more than n children. The tree in Figure 11-38 is an n-ary tree with $n = 3$. You can, of course, use the implementation just described for an n-ary tree. However, because you know the maximum number of children for each node, you can let each node reference its children directly. Figure 11-41 illustrates such a representation for the tree in Figure 11-38. This tree is shorter than the tree in Figure 11-40.

Exercise 35 at the end of this chapter discusses general trees further.

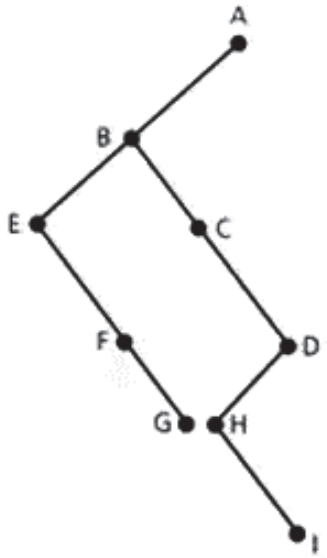


FIGURE 11-40
The binary tree that Figure 11-39 represents

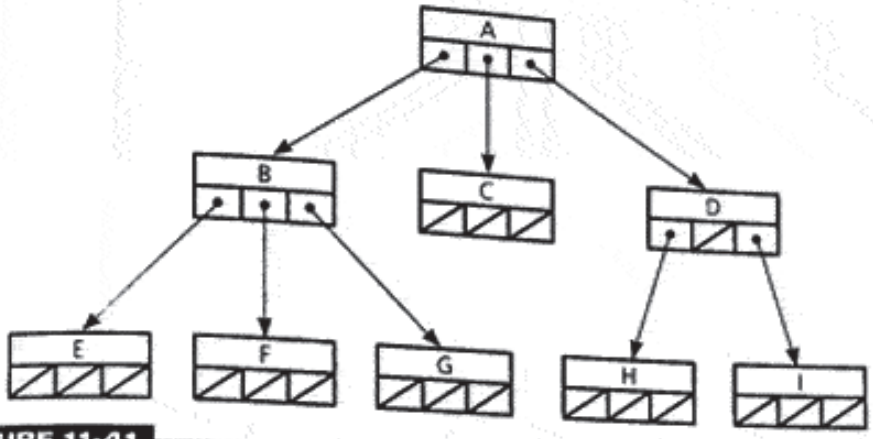


FIGURE 11-41
An implementation of the *n*-ary tree in Figure 11-38

Summary

1. Binary trees provide a hierarchical organization of data, which is important in many applications.
2. The implementation of a binary tree is usually reference based. If the binary tree is complete, an efficient array-based implementation is possible.
3. Traversing a tree is a useful operation. Intuitively, traversing a tree means to visit every node in the tree. Because the meaning of "visit" is application dependent, the traversal operations are implemented using an iterator.
4. The binary search tree allows you to use a binary search–like algorithm to search for an item with a specified value.
5. Binary search trees come in many shapes. The height of a binary search tree with n nodes can range from a minimum of $\lceil \log_2(n+1) \rceil$ to a maximum of n. The shape of a binary search tree determines the efficiency of its operations. The closer a binary search tree is to a balanced tree (and the farther it is from a linear structure), the closer the behavior of the *search* algorithm will be to a binary search (and the farther it will be from the behavior of a linear search).
6. An inorder traversal of a binary search tree visits the tree's nodes in sorted search-key order.
7. The treesort algorithm efficiently sorts an array by using the binary search tree's insertion and traversal operations.
8. If you save a binary search tree's data in a file while performing an inorder traversal of its nodes, you can restore the tree as a binary search tree of minimum height. If you save a binary search tree's data in a file while performing a preorder traversal of its nodes, you can restore the tree to its original form.

Cautions

1. If you use an array-based implementation of a complete binary tree, you must be sure that the tree remains complete as a result of insertions or deletions.
2. Operations on a binary search tree can be quite efficient. In the worst case, however—when the tree approaches a linear shape—the performance of its operations degrades and is comparable to that of a linear linked list. If you must avoid such a situation for a given application, you should use the balancing methods presented in Chapter 13.

Self-Test Exercises

1. Consider the tree in Figure 11-42. What node or nodes are
   a. The tree's root
   b. Parents
   c. Children of the parents in Part *b*
   d. Siblings
   e. Ancestors of 60
   f. Descendants of 70
   g. Leaves
2. What are the levels of all nodes in the tree in
   a. Figure 11-6b
   b. Figure 11-6c
3. What is the height of the tree in Figure 11-42?

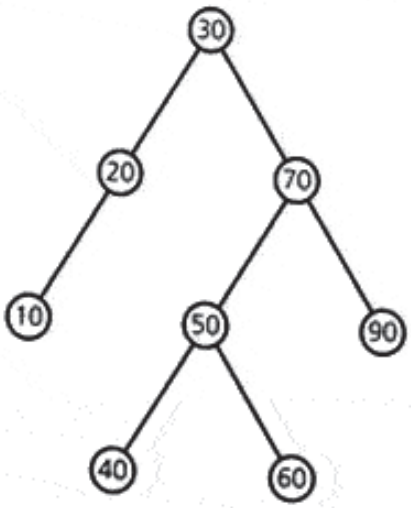


FIGURE 11-42
A tree for Self-Test Exercises 1, 3, 7, and 11 and for Exercises 7 and 14

4 Consider the binary trees in Figure 11-31. Which are complete? Which are full? Which are balanced?

5 What are the preorder, inorder, and postorder traversals of the binary tree in Figure 11-6a?

6 Beginning with an empty binary search tree, what binary search tree is formed when you insert the following values in the order given: J, N, B, A, W, E, T?

7 Starting with an empty binary search tree, in what order should you insert items to get the binary search tree in Figure 11-42?

8 Represent the full binary tree in Figure 11-36 with an array.

9 What complete binary tree does the array in Figure 11-43 represent?

10 Is the tree in Figure 11-44 a binary search tree?

11 Using the tree in Figure 11-42, trace the algorithm that searches a binary search tree, given a search key of

a. 50

b. 80

In each case, list the nodes in the order in which the search visits them.

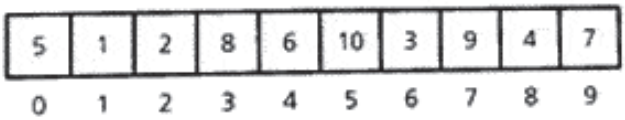


FIGURE 11-43
An array for Self-Test Exercise 9

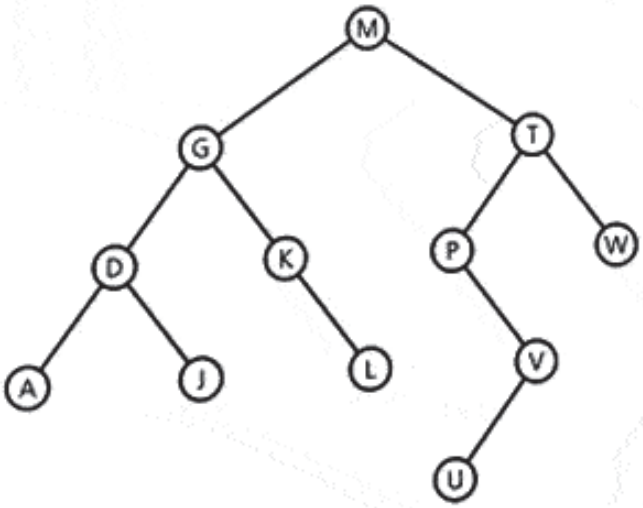


FIGURE 11-44
A tree for Self-Test Exercise 10 and for Exercise 2a

12. Trace the treesort algorithm as it sorts the following array into ascending order: 20 80 40 25 60 30.
13. a. What binary search tree results when you execute `readTree` with a file of the six integers 2, 4, 6, 8, 10, 12?

    b. Is the resulting tree's height a minimum? Is the tree complete? Is it full?

Exercises

1. Write pre- and postconditions for the operations of the ADT binary tree.
2. What are the preorder, inorder, and postorder traversals of the binary trees in

   a. Figure 11-44 b. Figure 11-6b c. Figure 11-6c

3. Consider the binary search tree in Figure 11-45. The numbers simply label the nodes so that you can reference them; they do not indicate the contents of the nodes.

   a. Which node must contain the inorder successor of the value in the root? Explain.

   b. In what order will an inorder traversal visit the nodes of this tree? Indicate this order by listing the labels of the nodes in the order that they are visited.

4. Beginning with an empty binary search tree, what binary search tree is formed when you insert the following values in the order given?

   a. T, N, W, J, E, A, B

   b. J, A, N, E, T, B, W

   c. J, T, E, N, B, A, W

5. Draw the complete binary tree that is formed when the following values are inserted in the order given: 3, 12, 5, 16, 6, 9.

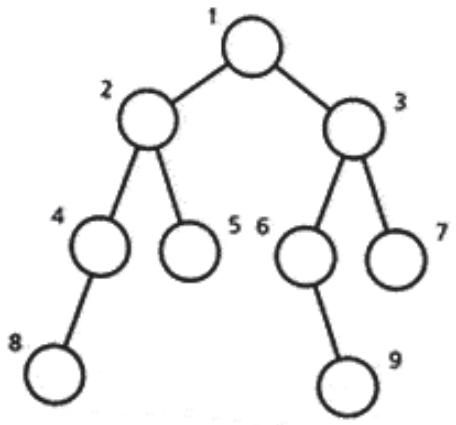


FIGURE 11-45
A binary search tree for Exercise 3

6. Arrange nodes that contain the letters D, Q, X, F, S, G, and W into two binary search trees: one that has maximum height and one that has minimum height.

7. Consider the binary search tree in Figure 11-42.

   a. What tree results after you insert the nodes 80, 65, 75, 45, 35, and 25, in that order?

   b. After inserting the nodes mentioned in Part *a*, what tree results when you delete the nodes 50 and 20?

8. a. What is the maximum number of nodes in a binary tree with 8 levels?

   b. What is the maximum and minimum number of levels of a tree with 2,011 nodes?

   c. What is the maximum and minimum number of leaves in a tree with 10 levels?

9. Given the following data: 10, 15, 5, 18, 14, 6, 20, 9

   a. What binary search tree is created by inserting the data in the order given?

   b. Given a search key of 12, trace the algorithm that searches the binary search tree that you created in Part *a*. List the nodes in the order in which the search visits them.

10. If you delete an item from a binary search tree and then insert it back into the tree, will you ever change the shape of the tree?

11. Given the ADT binary tree operations as defined in this chapter, what tree or trees does the following sequence of statements produce?

```
public void Ex11() {
  BinaryTree<Integer> t1 = new BinaryTree<Integer>(2);
  t1.attachLeft(5);

  BinaryTree<Integer> t2 = new BinaryTree<Integer>(8);
  t2.attachLeft(6);
  t2.attachRight(7);

  t1.attachRightSubtree(t2);

  BinaryTree<Integer> t3 = new BinaryTree<Integer>(7);
  t2.attachLeft(3);
  t2.attachRight(1);

  BinaryTree<Integer> t4 = new BinaryTree<Integer>(1, t1, t3);
} // end Ex11
```

12. Consider a method *isLeaf()* that returns `true` if an instance of *BinaryTree* is a one-node tree—that is, if it consists of only a leaf—and returns `false` otherwise.

   a. Add the method of *isLeaf* to *BinaryTree* so that the method is available to clients of the class.

   b. If *isLeaf* were not a member of *BinaryTree*, would a client of the class be able to implement *isLeaf*? Explain.

13. The operation

```
replace(in replacementItem:TreeItemType):boolean
```

locates, if possible, the item in a binary search tree with the same search key as `replacementItem`. If the tree contains such an item, `replace` replaces it with `replacementItem`. Thus, the fields of the original item are updated.

a. Add the operation `replace` to the reference-based implementation of the ADT binary search tree given in this chapter. The operation should replace an item without altering the tree structure.

b. Instead of adding `replace` as an operation of the ADT binary search tree, implement it as a client of `BinarySearchTree`. Will the shape of the binary tree remain the same?

14. Suppose that you traverse the binary search tree in Figure 11-42 and write the data item in each node visited to a file. You plan to read this file later and create a new binary search tree by using the ADT binary search tree operation `insert`. In creating the file, in what order should you traverse the tree so that the new tree will have exactly the same shape and nodes as the original tree? What does the file look like after the original tree is traversed?

15. Consider an array-based implementation of a binary search tree `bst`. Figure 11-11 presents such a representation for a particular binary search tree.

a. Depict the array in an array-based implementation for the binary search tree in Figure 11-20a.

b. Show the effect of each of the following sequential operations on the array in Part *a* of this exercise. For simplicity, assume that tree items are names.

```
bst.insert(new Name("Doug"));
bst.delete(new Name("Karen"));
bst.delete(new Name("Andrew"));
bst.insert(new Name("Sarah"));
```

c. Repeat Parts *a* and *b* of this exercise for the tree in Figure 11-21b.

d. Write an inorder traversal algorithm for this array-based implementation.

16. Duplicates in an ADT could mean either identical items or, more subtly, items that have identical search keys but with differences in other fields. If duplicates are allowed in a binary search tree, it is important to have a convention that determines the relationship between the duplicates. Items that duplicate the root of a tree should either all be in the left subtree or all be in the right subtree, and, of course, this property must hold for every subtree.

a. Why is this convention critical to the effective use of the binary search tree?

b. This chapter stated that you can delete an item from a binary search tree by replacing it with the item whose search key either immediately follows or immediately precedes the search key of the item to be deleted. If duplicates are allowed, however, the choice between inorder successor

inorder predecessor is no longer arbitrary. How does the convention of putting duplicates in either the left or right subtree affect this choice?

17. Complete the trace of the nonrecursive inorder traversal algorithm that Figure 11-15 began. Show the contents of the implicit stack as the traversal progresses.

18. Implement in Java the nonrecursive inorder traversal algorithm for a binary tree that was presented in this chapter. (See Exercise 17.)

19. Given the recursive nature of a binary tree, a good strategy for writing a Java method that operates on a binary tree is often first to write a recursive definition of the task. Given such a recursive definition, a Java implementation is often straightforward.

Write recursive definitions that perform the following tasks on arbitrary binary trees. Implement the definitions in Java. Must your methods be members of `BinaryTree`? For simplicity, assume that each data item in the tree is an `integer` object and that there are no duplicates.

a. Count the number of nodes in the tree. (*Hint*: If the tree is empty, the count is 0. If the tree is not empty, the count is 1 plus the number of nodes in the root's left subtree plus the number of nodes in the root's right subtree.)

b. Compute the height of a tree.

c. Find the maximum element.

d. Find the sum of the elements.

e. Find the average of the elements.

f. Find a specific item.

g. Determine whether one item is an ancestor of another (that is, whether one item is in the subtree rooted at the other item).

h. Determine the highest level that is full or, equivalently, has the maximum number of nodes for that level. (See Exercise 25.)

20. Consider a nonempty binary tree with two types of nodes: **min** nodes and **max** nodes. Each node has an integer value initially associated with it. You can define the **value** of such a **minimax** tree as follows:

- If the root is a min node, the value of the tree is equal to the ***minimum*** of
  - The integer stored in the root
  - The value of the left subtree, but only if it is nonempty
  - The value of the right subtree, but only if it is nonempty
- If the root is a max node, the value of the tree is equal to the ***maximum*** of the above three values.

Figure 11-46a shows a completed minimax tree.

a. Compute the value of the minimax tree in Figure 11-46. Each node is labeled with its initial value.

b. Write a general solution in Java for representing and evaluating these trees.

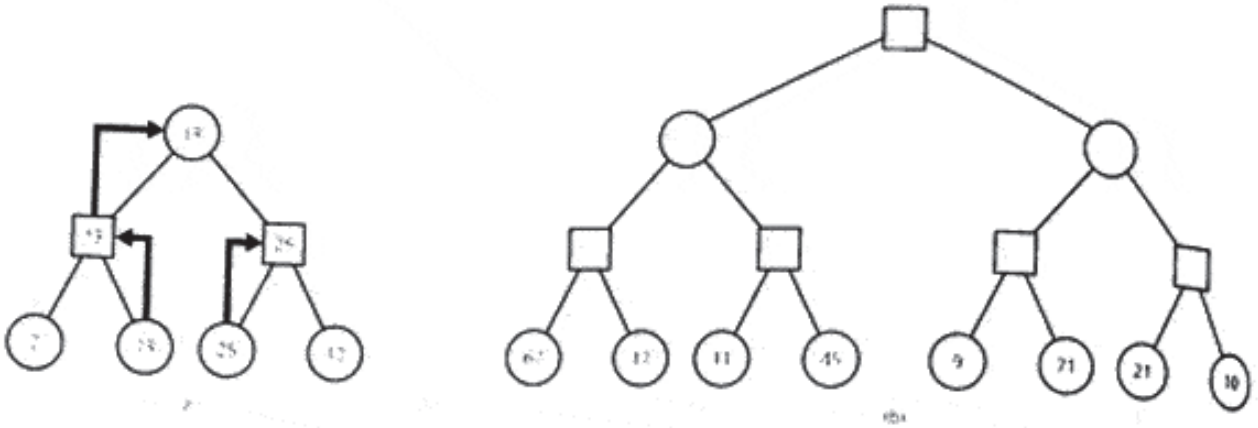


FIGURE 11-46
A minimax tree for Exercise 20

* 21. A binary search tree with a given set of data items can have several different structures that conform to the definition of a binary search tree. If you are given a list of data items, does at least one binary search tree whose preorder traversal matches the order of the items on your list always exist? Is there ever more than one binary search tree that has the given preorder traversal?

* 22. How many differently shaped, n-node binary trees are possible? How many differently shaped, n-node binary search trees are possible? (Write recursive definitions.)

23. Write pseudocode for a method that performs a range query for a binary search tree. That is, the method should visit all items that have a search key in a given range of values (such as all values between 100 and 1,000).

24. Prove Theorems 11-2 and 11-3 by using induction.

25. What is the maximum number of nodes that a binary tree can have at level n? Prove your answer by using induction. Use this fact to do the following:

    a. Rewrite the formal definition of a complete tree of height h.

    b. Derive a closed form for the formula

$$\sum_{i=1}^{h} 2^{i-1}$$

    What is the significance of this sum?

26. Prove by induction that a binary tree with n nodes has exactly $n + 1$ empty subtrees (or, in Java terms, $n + 1$ *null* references).

27. A binary tree is **strictly binary** if every nonleaf node has exactly two children. Prove by induction on the number of leaves that a strictly binary tree with n leaves has exactly $2n - 1$ nodes.

28. Consider two algorithms for traversing a binary tree. Both are nonrecursive algorithms that use an extra ADT for bookkeeping. Both algorithms have the following basic form:

```
Put the root of the tree in the ADT
while (the ADT is not empty) {
  Remove a node from the ADT and call it n
  Visit n
  if (n has a left child) {
    Put the child in the ADT
  }  // end if
  if (n has a right child) {
    Put the child in the ADT
  }  // end if
}  // end while
```

The difference between the two algorithms is the method for choosing a node *n* to remove from the ADT.

Algorithm 1: Remove the newest (most recently added) node from the ADT.

Algorithm 2: Remove the oldest (earliest added) node from the ADT.

a. In what order would each algorithm visit the nodes of the tree in Figure 11-19?

b. For each algorithm, describe an appropriate ADT for doing the bookkeeping. What should the ADT data be? Do not use extra memory unnecessarily for the bookkeeping ADT. Also, note that the traversal of a tree should not alter the tree in any way.

29. Describe how to save a binary tree in a file so that you can later restore the tree to its original shape. Compare the efficiencies of saving and restoring a binary tree and a binary search tree.

30. Design another algorithm to delete nodes from a binary search tree. This algorithm differs from the one described in this chapter when the node *N* has two children. First let *N*'s right child take the place of the deleted node *N* in the same manner in which you delete a node with one child. Next reconnect *N*'s left child (along with its subtree, if any) to the left side of the node containing the inorder successor of the search key in *N*.

31. Write iterative methods to perform insertion and deletion operations on a binary search tree.

32. Use inheritance to derive the class `BSTTreeIterator` from the class `TreeIterator` and implement the method `remove`. Is the traversal affected when a node of the tree is removed? Does this depend on the implementation of the iterator?

33. If you know in advance that you often access a given item in a binary search tree several times in a row before accessing a different item, you will search for the same item repeatedly. One way to address this problem is to add an extra bookkeeping component to your implementation. That is, you can maintain a last-accessed reference that will always reference the last item that any binary search tree operation accessed. Whenever you perform such an operation, you can check the search key of the item most recently accessed before performing the operation.

Revise the implementation of the binary search tree to add this new feature by adding the data field `lastAccessed` to the class.

34. The motivation for using a doubly linked list is the need to locate and delete a node in a list without traversing the list. The analogy for a binary search tree is to maintain parent references. That is, every node except the root will have a reference to its parent in the tree. Write insertion and deletion operations for this tree.

35. A node in a general tree, such as the one in Figure 11-38, can have an arbitrary number of children.

    a. Describe a Java implementation of a general tree in which every node contains an array of child references. Write a recursive preorder traversal method for this implementation. What are the advantages and disadvantages of this implementation?

    b. Consider the implementation of a general tree that this chapter described. Each node has two references: The left one references the node's oldest child and the right one references the node's next sibling. Write a recursive preorder traversal method for this implementation.

    c. Every node in a binary tree T has at most two children. Compare the oldest-child/next-sibling representation of T described in Part *b* to the left-child/right-child representation of a binary tree described in this chapter. Does one representation simplify the implementation of the ADT operations? Are the two representations ever the same?

36. The section "Saving a Binary Search Tree in a File" mentions that you can save a binary search tree in a file by adding the interface `java.io.Serializable` to the various classes involved in the implementation of the tree. Name all of these classes when the binary search tree contains instances of the class `Person` presented in this chapter.

Programming Problems

1. Write an array-based implementation of the ADT binary search tree that uses dynamic memory allocation. Use a data structure like the one in Figure 11-11.

2. Repeat Programming Problem 1 for complete binary trees.

3. Write a Java program that learns about a universe of your choice by asking the user yes/no questions. For example, your program might learn about animals by having the following dialogue with its user. (User responses are in uppercase.)

```
Think of an animal and I will guess it.
Does it have legs? YES
Is it a cat? YES
I win! Continue? YES
```

```
Please type a question whose answer is yes for an
earthworm and no for a snake:
DOES IT LIVE UNDERGROUND?
Continue? YES

Think of an animal and I will guess it.
Does it have legs? NO
Does it live underground? NO
Is it a snake? NO
I give up. What is it? FISH
Please type a question whose answer is yes for a
fish and no for a snake:
DOES IT LIVE IN WATER?
Continue? NO

Good-bye.
```

The program begins with minimal knowledge about animals: It knows that cats have legs and snakes do not. When the program incorrectly guesses "snake" the next time, it asks for the answer and also asks for a way to distinguish between snakes and earthworms.

The program builds a binary tree of questions and animals. A YES response to a question is stored in the question's left child; a NO response is stored in the question's right child.

4. Write a program that maintains the names, addresses, and telephone numbers of your friends and relatives and thus serves as an address book. You should be able to enter, delete, modify, or search this data. The person's name should be the search key, and initially you can assume that the names are unique. The program should be able to save the address book in a file for use later.

   Design a class to represent the people in the address book and another class to represent the address book itself. This class should contain a binary search tree of people as a data field.

   You can enhance this problem by adding birth dates to the database and by adding an operation that lists everyone who satisfies a given criterion. For example, it might list people born in a given month or people who live in a given state. You should also be able to list everyone in the database.

5. Write a program that provides a way for you to store and retrieve telephone numbers. Design a user interface that provides the following operations:

   Add: Adds a person's name and phone number to the phone book.

   Delete: Deletes a given person's name and phone number from the phone book, given only the name.

   Find: Locates a person's phone number, given only the person's name.

   Change: Changes a person's phone number, given the person's name and new phone number.

   Quit: Quits the application, after first saving the phone book in a text file.

   You can proceed as follows:

   - Design and implement the class `Person`, which represents the name and phone number of a person. You will store instances of this class in the phone book.

- Design and implement the class `PhoneBook`, which represents the phone book. The class should contain a binary search tree as a data field. This tree contains the people in the book.
- Add methods that use a text file to save and restore the tree.
- Design and implement the class `Menu`, which provides the program's user interface.

The program should read data from a text file when it begins and save data into a text file when the user quits the program.

6. In this chapter, a single iterator class was created to perform any one of the three binary tree traversals. The user could specify which of the traversals to use by calling `setPreorder`, `setInorder`, or `setPostorder`. An alternative implementation is based on creating a separate iterator class for each of the traversals. Also, as was mentioned in this chapter, the space and time requirements of the traversal can be minimized if the traversals are based on nonrecursive algorithms.

   For example, the following pseudocode demonstrates how the method `inorderTraverse` presented in this chapter can be modified slightly and called from the method `next` in the iterator. The original version of `inorderTraverse` used a queue to store the entire traversal. This version of the method will produce the next node in the traversal only as needed. It is therefore slightly different in that the statement that queued a node is replaced with a return of the node.

```
+inorderTraverse(in treeNode:TreeNode):TreeItemType
// Nonrecursively traverses a binary tree
// inorder.
  curr = treeNode   // start at treeNode
  done = false
  while (!done) {
    if (curr != null) {
      visitStack.push(curr)
      // traverse the left subtree
      curr = curr.leftChild
    }
    else {
      if (!visitStack.isEmpty()) {
        curr = visitStack.pop()
        return curr.item
      }
      else {
        done = true
      }  // end if
    }  // end if
  }  // end while
```

Implement an iterator class for each of the three traversals using nonrecursive methods such that the storage requirements are never greater than O(*height of the tree*).