The no-parameter constructor merely sets the data field `root` to **`null`**.

```
public BinaryTree() {
    root = null;
}
```

The constructor that takes a `Node` as a parameter is a protected constructor. This is because client classes do not know about the `Node` class. This constructor can be used only by methods internal to the `BinaryTree` class and its subclasses.

```
protected BinaryTree(Node<E> root) {
    this.root = root;
}
```

The third constructor takes three parameters: data to be referenced by the root node and two binary trees that will become its left and right subtrees.

```
/** Constructs a new binary tree with data in its root leftTree
    as its left subtree and rightTree as its right subtree.
 */
public BinaryTree(E data, BinaryTree<E> leftTree,
                  BinaryTree<E> rightTree) {
    root = new Node<>(data);
    if (leftTree != null) {
        root.left = leftTree.root;
    } else {
        root.left = null;
    }
    if (rightTree != null) {
        root.right = rightTree.root;
    } else {
        root.right = null;
    }
}
```

If `leftTree` is not **`null`**, the statement

```
root.left = leftTree.root;
```

executes. After its execution, the root node of the tree referenced by `leftTree` (`leftTree.root`) is referenced by `root.left`, making `leftTree` the left subtree of the new root node. If `lT` and `rT` are type `BinaryTree<Character>` and `lT.root` references the root node of binary tree `x + y` and `rT.root` references the root node of binary tree `a / b`, the statement

```
BinaryTree<Character> bT = new BinaryTree<> ('*', lT, rT);
```

would cause `bT` to reference the tree shown in Figure 6.12.

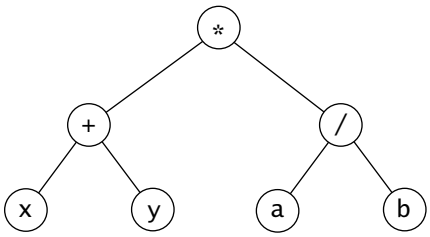


FIGURE 6.12
The Expression Tree
`(x + y) * (a / b)`

The `getLeftSubtree` and `getRightSubtree` Methods

The `getLeftSubtree` method returns a binary tree whose root is the left subtree of the object on which the method is called. It uses the protected constructor discussed before to

construct a new BinaryTree<E> object whose root references the left subtree of this tree. The getRightSubtree method is symmetric.

```
/** Return the left subtree.
    @return The left subtree or null if either the root or
            the left subtree is null
 */
public BinaryTree<E> getLeftSubtree() {
    if (root != null && root.left != null) {
        return new BinaryTree<>(root.left);
    } else {
        return null;
    }
}
```

The isLeaf Method

The isLeaf method tests to see whether this tree has any subtrees. If there are no subtrees, then true is returned.

```
/** Determine whether this tree is a leaf.
    @return true if the root has no children
 */
public boolean isLeaf() {
    return (root.left == null && root.right == null);
}
```

The toString Method

The toString method generates a string representation of the BinaryTree for display purposes. The string representation is a preorder traversal in which each local root is indented a distance proportional to its depth. If a subtree is empty, the string "null" is displayed. The tree in Figure 6.12 would be displayed as follows:

```
*
  +
    x
      null
      null
    y
      null
      null
  /
    a
      null
      null
    b
      null
      null
```

The toString method creates a StringBuilder and then calls the recursive toString method (described next) passing the root and 1 (depth of root node) as arguments.

```
public String toString() {
    var sb = new StringBuilder();
    toString(root, 1, sb);
    return sb.toString();
}
```

The Recursive toString Method

This method follows the preorder traversal algorithm given in Section 6.2. It begins by appending a string of spaces proportional to the level so that all nodes at a particular level

will be indented to the same point in the tree display. Then, if the node is `null`, the string `"null\n"` is appended to the `StringBuilder`. Otherwise the string representation of the node is appended to the `StringBuilder` and the method is recursively called on the left and right subtrees.

```
/** Converts a subtree to a string.
    Performs a preorder traversal.
    @param node The local root
    @param depth The depth
    @param sb The StringBuilder to save the output
 */
private void toString(Node<E> node, int depth,
                      StringBuilder sb) {
    for (int i = 1; i < depth; i++) {
        sb.append(" ");
    }
    if (node == null) {
        sb.append("null\n");
    } else {
        sb.append(node.toString());
        sb.append("\n");
        toString(node.left, depth + 1, sb);
        toString(node.right, depth + 1, sb);
    }
}
```

Reading a Binary Tree

If we use a `Scanner` to read the individual lines created by the `toString` method previously discussed, we can reconstruct the binary tree using the algorithm:

1. Read a line that represents information at the root.
2. Remove the leading and trailing spaces using the `String.trim` method.
3. **`if`** it is **`"null"`**
4. Return **`null`**.

   `else`
5. Recursively read the left child.
6. Recursively read the right child.
7. Return a tree consisting of the root and the two children.

The tree that is constructed will be type `BinaryTree<String>`. The code for a method that implements this algorithm is shown in Listing 6.2.

LISTING 6.2

Method to Read a Binary Tree

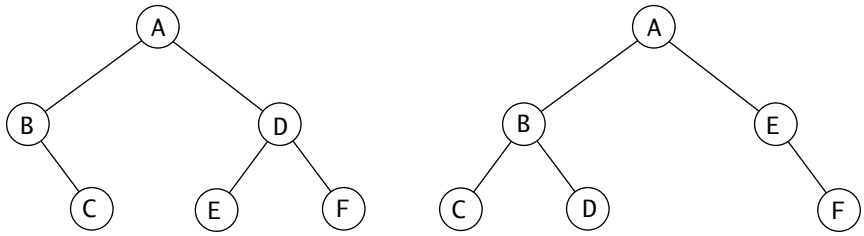
```
/** Method to read a binary tree.
    pre: The input consists of a preorder traversal
         of the binary tree. The line "null" indicates a null tree.
    @param scan the Scanner attached to the input file.
    @return The binary tree
 */
public static BinaryTree<String> readBinaryTree(Scanner scan) {
    // Read a line and trim leading and trailing spaces.
    String data = scan.nextLine().trim();
    if (data.equals("null")) {
        return null;
    } else {
        BinaryTree<String> leftTree = readBinaryTree(scan);
```

```
            BinaryTree<String> rightTree = readBinaryTree(scan);
            return new BinaryTree<>(data, leftTree, rightTree);
        }
    }
```

EXERCISES FOR SECTION 6.3

SELF-CHECK

1. Draw the linked representation of the following two trees.



2. Show the tree that would be built by the following input string:

```
30
  15
    4
      null
      null
    20
      18
        null
        19
          null
          null
      null
  35
    32
      null
      null
    38
      null
      null
```

3. What can you say about this tree?

4. Write the strings that would be displayed for the two binary trees in Figure 6.6.

PROGRAMMING

1. Write a method for the `BinaryTree` class that returns the preorder traversal of a binary tree as a sequence of strings each separated by a space.

2. Write a method to display the postorder traversal of a binary tree in the same form as Programming Exercise 1.

3. Write a method to display the inorder traversal of a binary tree in the same form as Programming Exercise 1, except place a left parenthesis before each subtree and a right parenthesis after each subtree. Don't display anything for an empty subtree. For example, the expression tree shown in Figure 6.12 would be represented as `(((x) + (y)) * ((a) / (b)))`.

6.4 Lambda Expressions and Functional Interfaces

Java 8 introduced new features that enable functional programming. In *functional programming*, you can assign functions (methods) to variables or pass them as arguments to another function. The behavior of the function called with a function argument will vary, depending on its function argument. Assume you wrote a function called `plot` that cycled through angles from 0 to 360° in 5° increments and produced a graph showing a particular function value for each angle. If a method `plot` took a function argument, you could pass it a function such as `sin` or `cos`. If you passed it `sin`, function plot would show a graph of sine values; if you passed it `cos`, it would show a graph of cosine values.

Java can't really pass methods as arguments, but it accomplishes the same thing using lambda expressions and functional interfaces. A *lambda expression* is a method without a declaration—sometimes called an *anonymous method* because it doesn't have a name. A lambda expression is a shorthand notation that allows you to write a method where you are going to use it. This is useful when a method is going to be used only once and it saves you the effort of writing a separate method declaration. A lambda expression consists of a parameter list and either an expression or a statement block separated by the symbol ->.

EXAMPLE 6.2 The following lambda expression has no arguments as indicated by the empty parentheses. It simply displays the message shown in the `println`.

```
() -> System.out.println("That's all folks") // displaysmessage
```

EXAMPLE 6.3 The next lambda expression has the value of `m` cubed. Because it has a single untyped argument `m`, the parentheses are omitted.

```
m -> m * m * m
```

EXAMPLE 6.4 The next lambda expression represents a method that returns the larger of its two arguments. Because the method body has multiple statements, it is enclosed in braces. The argument types could be omitted but the parentheses are required.

```
(int x, int y) -> {
    if (x >= y)
       return x;
    return y;
}
```

We will see how to execute lambda expressions (anonymous methods) in the next section.

SYNTAX Lambda Expression

FORM:

(*parameter list*) -> *expression*

or

(*parameter list*) -> {*statement list*}

EXAMPLE:

```
x -> x * x
(d, e) -> {
    sb.append(d.toString());
    sb.append(" : ");
    sb.append(e.toString());
}
```

INTERPRETATION:

The *parameter list* for the anonymous method has the same syntax as the *parameter list* for a method—a comma separated list of type identifier sequences enclosed in parentheses. If the compiler can infer the types, they can be omitted. If there is only one parameter, then the parentheses may also be omitted. An empty *parameter list* is denoted by `()`. The method body (following the `->`) may be an expression, a single statement, or a statement block enclosed in curly braces.

Functional Interfaces

To cause a lambda expression to be executed, we can pass it as an argument to another method. The data type of the parameter corresponding to a lambda expression must be a functional interface. A *functional interface* is an interface that declares exactly one abstract method. Java provides a set of functional interfaces that take lambda expressions as arguments.

Table 6.2 lists some of the functional interfaces in the `java.util.function` package. The type parameters `T` and `U` represent input types and `R` the result type. Each functional interface has a single abstract method. The abstract methods described in the table are `accept`, `apply`, `compare`, and `test`.

TABLE 6.2
Selected Java Functional Interfaces

Interface	Abstract Method	Description
`BiConsumer<T, U>`	`void accept(T t, U u)`	Represents an operation that accepts two input arguments and returns no result
`BiFunction<T, U, R>`	`R apply(T t, U u)`	Represents an operation that accepts two arguments of different types and produces a result
`BinaryOperator<T>`	`T apply(T t1, T t2)`	Represents an operation upon two operands of the same type, producing a result of the operand type
`Comparator<T>`	`int compare(T t1, T t2)`	Represents an operation that accepts two arguments of the same type and returns a positive, zero, or negative result based on their ordering (negative if `t1 < t2`, zero if `t1` equals `t2`, positive if `t1 > t2`)
`Consumer<T>`	`void accept(T t)`	Represents an operation that accepts one input argument and returns no result
`Function<T, R>`	`R apply(T t)`	Represents a function that accepts one argument and produces a result
`Predicate<T>`	`boolean test(T t)`	Represents a predicate (boolean-valued function) of one argument

EXAMPLE 6.5 In the following example, we use the functional interface `BinaryOperator`. As described in Table 6.2, it has a single abstract method `apply` that accepts two `Integer` arguments and returns an `Integer` result.

Listing 6.3 shows a class that uses the lambda expression from Example 6.4 that returns the larger of its two arguments. The statement

```
BinaryOperator<Integer> bO = (x, y) -> {
                         if (x > y)
                             return x;
                         return y;
};
```

creates an object of an anonymous class type that implements interface `Binary Operator`. The statement block to the right of -> implements method `apply`. Therefore, the statement

```
System.out.println("The larger number is : " + bO.apply(m, n));
```

causes the statement block above to execute with `m` and `n` as its arguments. The larger of the two data values entered will be returned as the method result and displayed.

LISTING 6.3

Using Functional Interface `BinaryOperator` with a Lambda Expression

```
import java.util.function.BinaryOperator;
import java.util.Scanner;

public class TestBinaryOperator {
    public static void main(String[] args) {
       var sc = new Scanner(System.in);
       System.out.println("Enter 2 integers: ");
       int m = sc.nextInt();
       int n = sc.nextInt();

       BinaryOperator<Integer> bO = (x, y) -> {
                                    if (x > y)
                                        return x;
                                    return y;
       };

       System.out.println("The larger number is : " + bO.apply(m,n));
    }
}
```

Custom Functional Interfaces

In addition to the functional interfaces provided by Java, library writers may define their own. For example, in Section 3.4, which discusses JUnit5, we used a lambda expression in the statement

```
assertThrows(NullPointerException.class,
        ()-> ArraySearch.search(y, 10));
```

Here the lambda expression passes to `assertThrows` the code (an array search) that is expected to throw a `NullPointerException` when the array `y` is empty.

The `assertThrows` method has the following signature:

```
public static <T extends Throwable>
assertThrows(Class<T> expected, Executable executable)
```

`Executable` is a custom functional interface that takes a lambda expression that defines a function taking no arguments and returning `void`.

Passing a Lambda Expression as a Function Argument

We started our discussion of lambda expressions by stating that we wanted to be able to write a method that could plot the values of another function passed to it. The plot would vary depending on the function argument. We can accomplish this by passing a lambda expression that implements the Functional Interface as the method argument.

EXAMPLE 6.6 The following method displays the values of a function (its last argument) in the range represented by `low` to `high` in increments of step. The function represented by `f` takes an `Integer` argument and returns a `Double` result.

```
/** Displays values associated with function f in
 *  the range specified.
 *  @param low the lower bound
 *  @param high the upper bound
 *  @param step the increment
 *  @param f the function object
 */
public static void show(int low, int high, int step,
                        Function<Integer, Double> f) {
    for (int i = low; i <= high; i += step) {
        System.out.println(i + " : " + f.apply(i));
    }
}
```

We can call function show using the statements

```
Function<Integer, Double> f;
f = angle -> Math.cos(Math.toRadians(angle));
show(0, 360, 30, f);
```

The first statement declares `f` to be an object of type `Function<Integer, Double>`. The second statement assigns `f` to a lambda expression with an `Integer` argument `angle`. The method body to the right of `->` implements abstract method `apply`. Therefore, `f.apply(angle)` in method `show` will calculate and return the cosine value of angle after first converting it to radians. The last statement calls `show`, passing it the range boundaries and increment and function `f`. The `for` loop body in method `show` will display a table of angle and cosine values for 0° through 360° in steps of 30°:

```
0 : 1.0
30 : 0.8660254037844387
60 : 0.5000000000000001
...
```

A more compact way of doing this would be to replace the three statements above that declare and initialize `f` and call `show` with

```
show(0, 360, 30, angle -> Math.cos(Math.toRadians(angle)));
```

This statement passes the lambda expression directly to method `show` as an anonymous `Function` object.

PITFALL

Passing a Lambda Expression to an Argument That Is Not a Functional Interface

In Listing 6.3, we declared `bO` as a lambda expression that implements the `BinaryOperator` functional interface. We might consider removing the declaration for `bO` and passing the lambda expression directly in the `println`:

```
System.out.println("The larger number is : " + () -> {
                                if (x > y)
                                    return x;
                                return y;
                            });
```

This would result in the error message `lambda expression not expected here`.

A General Preorder Traversal Method

The recursive `toString` performs a preorder traversal to generate the string representation of a tree. There are other applications that require a preorder traversal but may perform a different operation on each node other than displaying its string representation. Thus, we want to separate the traversal from the action performed on each node. We will specify the action in a lambda interface.

At each node, we need to know the current node and its depth to perform an action, so we will use a functional interface that has parameter types `Node<E>` for the current node and `Integer` for its depth. Table 6.2 shows functional interface `BiConsumer<T, U>` that takes two arguments and has an abstract method `accept` whose return type is `void`. We will specify the operation to be performed on each node in a lambda expression that instantiates this interface.

The preorder traversal starter method below has a single parameter `consumer`, which references the lambda expression that specifies the action to be performed on each node. It calls the recursive preorder traversal method passing it the root node, 1 for its depth, and the `BiConsumer` object referenced by `consumer`.

```
/** Starter method for preorder traversal
 *  @param consumer an object that instantiates
 *          the BiConsumer interface. Its method implements
 *          abstract method accept.
 */
public void preOrderTraverse(BiConsumer<E, Integer> consumer) {
    preOrderTraverse(root, 1, consumer);
}
```

The private `preOrderTraverse` method performs the actual traversal.

```
/**
 *  Performs a recursive pre-order traversal of the tree,
 *  applying the action specified in the consumer object.
 *  @param consumer object whose accept method specifies
 *          the action to be performed on each node
 */
private void preOrderTraverse(Node<E> node, int depth,
                              BiConsumer<E, Integer> consumer) {
    if (node == null) {
        consumer.accept(null, depth);
    } else {
```

```
            consumer.accept(node.data, depth);
            preOrderTraverse(node.left, depth + 1, consumer);
            preOrderTraverse(node.right, depth + 1, consumer);
        }
    }
```

Using preOrderTraverse

We can rewrite the `toString` method to use the `preOrderTraverse` method. The `preOrderTraverse` method visits each node in preorder applying the statement block specified in the lambda expression passed as an argument to the preorder traversal methods. The lambda expression passed by `toString` to `preOrderTraverse` has arguments representing the node (`e`) and its depth (`d`). When `preOrderTraverse` applies its statement block to a node, it creates a new `StringBuilder` object consisting of `d` blanks followed by the string representation of the current node and `"\n"`.

```
public String toString() {
    var sb = new StringBuilder();
    preOrderTraverse((e, d) -> {
        for (int i = 1; i < d; i++) {
             sb.append(" ");
        }
        sb.append(e);
        sb.append("\n");
    });
    return sb.toString();
}
```

EXERCISES FOR SECTION 6.4

SELF-CHECK

1. Describe the effect of each lambda expression below that is valid. Correct the expression if it is not a valid lambda expression and then describe its effect.
   a. `(int m, n) -> 2 * m + 3 * n`
   b. `(int m, double x) -> m * (int) x`
   c. `msg -> return "***" + msg + "***"`
   d. `System.out.println("Hello")`
   e.
```
(x, y) -> {System.out.println(x);
           System.out.println("________________");
           System.out.println(y);
           System.out.println("________________");
           System.out.println("________________");
  };
```
   f. `(x, y) -> Math.sqrt(x * x + y * y)`
   g. `(x) -> x > 0`
2. Select a functional interface appropriate for each of the expressions in Self-Check Exercise 1 from Table 6.2 or the `java.util.function` library.
3. Declare a function object of the correct type for each of the lambda expressions in Self-Check Exercise 1 and assign the lambda expression to it.

PROGRAMMING

1. Write a lambda expression that takes a double argument (x) and an integer argument (n). The method result should be the double value raised to the power n. Do this using a Java API method and also using a loop.
2. Write and test a Java class that enters two numbers and displays the result of calling each lambda expression in Programming Exercise 1. Also, compare the results to make sure that they are the same.
3. Modify the program in Example 6.6 to use two function objects that calculate trigonometric values and display the angle and corresponding values for each function object on the same line.
4. Write a generic `postOrderTraverse` method for the `BinaryTree` class similar to the `preOrderTraverse` method.
5. Write a generic `inOrderTraverse` method for the `BinaryTree` class similar to the `preOrderTraverse` method.

6.5 Binary Search Trees

Overview of a Binary Search Tree

In Section 6.1, we provided the following recursive definition of a binary search tree:

A set of nodes T is a binary search tree if either of the following is true:

- T is empty.
- If T is not empty, its root node has two subtrees, T_L and T_R, such that T_L and T_R are binary search trees and the value in the root node of T is greater than all values in T_L and is less than all values in T_R.

Figure 6.13 shows a binary search tree that contains integers. We can use the following algorithm to find an object in a binary search tree.

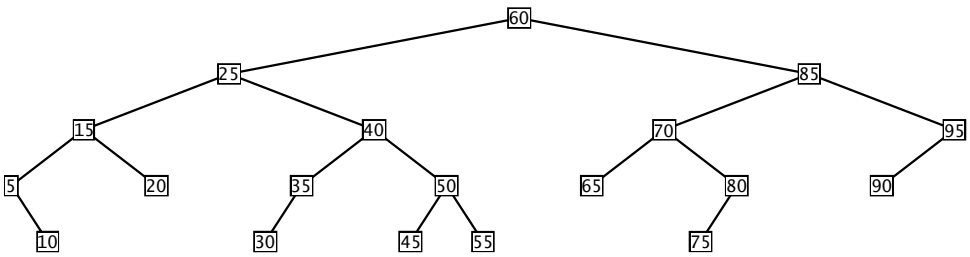


FIGURE 6.13 Binary Search Tree Containing Integers

Recursive Algorithm for Searching a Binary Search Tree

1. **`if`** the root is **`null`**
2. The item is not in the tree; return **`null`**.
3. Compare the value of `target`, the item being sought, with `root.data`.
4. **`if`** they are equal

5. `target` has been found, return the data at the root.

`else if` `target` is less than `root.data`

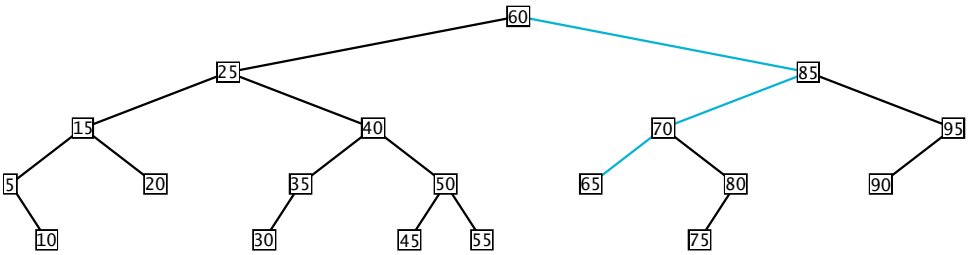
6. Return the result of searching the left subtree.

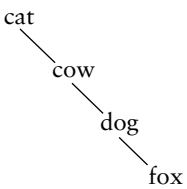
`else`

7. Return the result of searching the right subtree.

EXAMPLE 6.7 Suppose we wish to find *66* in Figure 6.13. We first compare *66* with *60*. Because *66* is greater than *60*, we continue the search with the right subtree and compare *66* with *85*. Because *66* is less than *85*, we continue with the left subtree and compare *66* with *70*. Because *66* is less than *70*, we continue with *65*. Now, *65* has no right child, and *66* is greater than *65*, so we conclude that *66* is not in this binary search tree. The path we followed is shown in color in Figure 6.14.

FIGURE 6.14
Searching for **66**





Performance

Searching the tree in Figure 6.14 is O(log*n*). However, if a tree is not very full, performance will be worse. The tree in the figure to the left has only right subtrees, so searching it is O(*n*).

Interface SearchTree

As described, the binary search tree is a data structure that enables efficient insertion, search, and retrieval of information (best case is O(log*n*)). Table 6.3 shows a `SearchTree<E>` interface for a class that implements the binary search tree. The interface includes methods for insertion (`add`), search (`boolean contains` and `E find`), and removal (`E delete` and `boolean remove`). Next, we discuss a class that implements this interface.

TABLE 6.3
The `SearchTree <E>` Interface

Method	Behavior
`boolean add(E item)`	Inserts `item` where it belongs in the tree. Returns **`true`** if `item` is inserted; **`false`** if it isn't (already in tree)
`boolean contains(E target)`	Returns **`true`** if `target` is found in the tree
`E find(E target)`	Returns a reference to the data in the node that is equal to `target`. If no such node is found, returns **`null`**
`E delete(E target)`	Removes `target` (if found) from tree and returns it; otherwise, returns **`null`**
`boolean remove(E target)`	Removes `target` (if found) from tree and returns **`true`**; otherwise, returns **`false`**

The BinarySearchTree Class

In the class `BinarySearchTree<E extends Comparable<E>>`, the type parameter `E` is specified when we create a new `BinarySearchTree` and `E` must implement the `Comparable` interface.

Table 6.4 shows the data fields declared in the class. These data fields are used to store a second result from the recursive `add` and `delete` methods that we will write for this class. Neither result can be returned directly from the recursive `add` or `delete` method because they return a reference to a tree node affected by the insertion or deletion operation. The interface for method `add` in Table 6.3 requires a **`boolean`** result (stored in `addReturn`) to indicate success or failure. Similarly, the interface for `delete` requires a type `E` result (stored in `deleteReturn`) that is either the item deleted or **`null`**.

The class heading and data field declarations follow. Note that class `BinarySearchTree` extends class `BinaryTree` and implements the `SearchTree` interface (see Figure 6.15). Besides the data fields shown, class `BinarySearchTree` inherits the data field `root` from class `BinaryTree` (declared as **`protected`**) and the inner class `Node<E>`.

```
public class BinarySearchTree<E extends Comparable<E>>
             extends BinaryTree<E> implements SearchTree<E> {
    // Data Fields
    /** Return value from the public add method. */
    protected boolean addReturn;
    /** Return value from the public delete method. */
    protected E deleteReturn;
    . . .
}
```

TABLE 6.4
Data Fields of Class `BinarySearchTree <E extends Comparable<E>>`

Data Field	Attribute
`protected boolean addReturn`	Stores a second return value from the recursive `add` method that indicates whether the item has been inserted
`protected E deleteReturn`	Stores a second return value from the recursive `delete` method that references the item that was removed from the tree

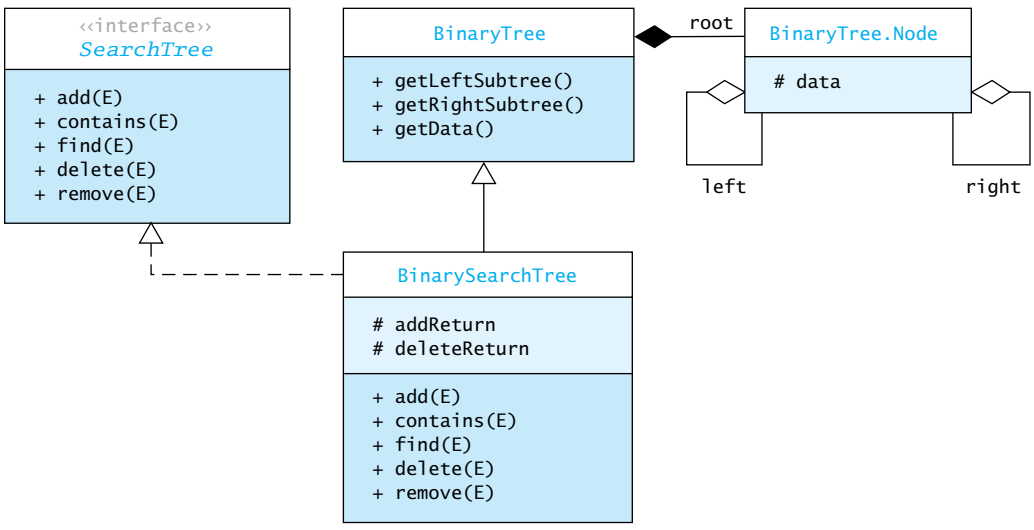


FIGURE 6.15
UML Diagram of `BinarySearchTree`

Implementing the `find` Methods

Earlier, we showed a recursive algorithm for searching a binary search tree. Next, we show how to implement this algorithm and a nonrecursive starter method for the algorithm. Our method `find` will return a reference to the node that contains the information we are seeking.

Listing 6.4 shows the code for method `find`. The starter method calls the recursive method with the tree root and the object being sought as its parameters. If `bST` is a reference to a `BinarySearchTree`, the method call `bST.find(target)` invokes the starter method.

The recursive method first tests the local root for **`null`**. If it is **`null`**, the object is not in the tree, so **`null`** is returned. If the local root is not **`null`**, the statement

```
int compResult = target.compareTo(localRoot.data);
```

compares `target` to the data at the local root. Recall that method `compareTo` returns an `int` value that is negative, zero, or positive depending on whether the object (`target`) is less than, equal to, or greater than the argument (`localRoot.data`).

If the objects are equal, we return the data at the local root. If `target` is smaller, we recursively call the method `find`, passing the left subtree root as the parameter.

```
return find(localRoot.left, target);
```

Otherwise, we call `find` to search the right subtree.

```
return find(localRoot.right, target);
```

LISTING 6.4

`BinarySearchTree` `find` Method

```
/** Starter method find.
    pre: The target object must implement
         the Comparable interface.
    @param target The Comparable object being sought
    @return The object, if found, otherwise null
 *
public E find(E target) {
    return find(root, target);
}

/** Recursive find method.
    @param localRoot The local subtree's root
    @param target The object being sought
    @return The object, if found, otherwise null
 */
private E find(Node<E> localRoot, E target) {
    if (localRoot == null)
        return null;

    // Compare the target with the data field at the root.
    int compResult = target.compareTo(localRoot.data);
    if (compResult == 0)
        return localRoot.data;
    else if (compResult < 0)
        return find(localRoot.left, target);
    else
        return find(localRoot.right, target);
}
```

Insertion into a Binary Search Tree

Inserting an item into a binary search tree follows a similar algorithm as searching for the item because we are trying to find where in the tree the item would be, if it were there. In searching, a result of **`null`** is an indicator of failure; in inserting, we replace this **`null`** with a new leaf that contains the new item. If we reach a node that contains the object we are trying to insert, then we can't insert it (duplicates are not allowed), so we return **`false`** to indicate that we were unable to perform the insertion. The insertion algorithm follows.

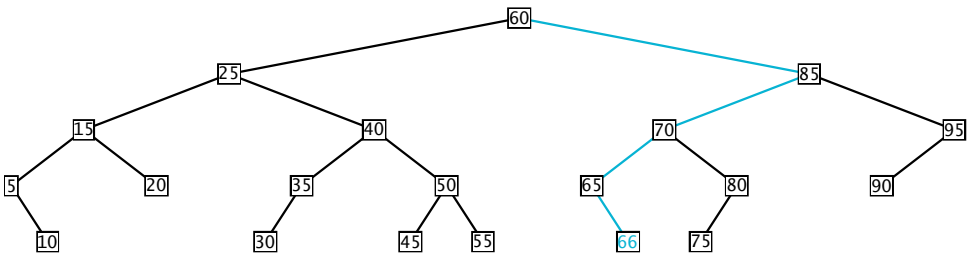
Recursive Algorithm for Insertion in a Binary Search Tree

1. **`if`** the root is **`null`**
2. Replace empty tree with a new tree with the item at the root and return **`true`**.
3. **`else if`** the item is equal to root.data
4. The item is already in the tree; return **`false`**.
5. **`else if`** the item is less than root.data
6. Recursively insert the item in the left subtree.
7. **`else`**
8. Recursively insert the item in the right subtree.

The algorithm returns **`true`** when the new object is inserted and **`false`** if it is a duplicate (the second stopping case). The first stopping case tests for an empty tree. If so, a new `BinarySearchTree` is created and the new item is stored in its root node (Step 2).

EXAMPLE 6.8 To insert *66* into Figure 6.13, we would follow the steps shown in Example 6.7 except that when we reached *65*, we would insert *66* as the right child of the node that contains *65* (see Figure 6.16).

FIGURE 6.16
Inserting 66



Implementing the add Methods

Listing 6.5 shows the code for the starter and recursive `add` methods. The recursive `add` follows the algorithm presented earlier, except that the return value is the new (sub)tree that contains the inserted item. The data field `addReturn` is set to **`true`** if the item is inserted and to **`false`** if the item already exists. The starter method calls the recursive method with the root as its argument. The root is set to the value returned by the recursive method (the modified tree). The value of `addReturn` is then returned to the caller.

In the recursive method, the statements

```
addReturn = true;
return new Node<>(item);
```

execute when a **`null`** branch is reached. The first statement sets the insertion result to **`true`**; the second returns a new node containing `item` as its data.

The statements

```
addReturn = false;
return localRoot;
```

execute when `item` is reached. The first statement sets the insertion result to **`false`**; the second returns a reference to the subtree that contains `item` in its root.

If `item` is less than the root's data, the statement

```
localRoot.left = add(localRoot.left, item);
```

attempts to insert `item` in the left subtree of the local root. After returning from the call, this left subtree is set to reference the modified subtree, or the original subtree if there is no insertion. The statement

```
localRoot.right = add(localRoot.right, item);
```

affects the right subtree of `localRoot` in a similar way.

LISTING 6.5

`BinarySearchTree` `add` Methods

```
/** Starter method add.
    pre: The object to insert must implement the
         Comparable interface.
    @param item The object being inserted
    @return true if the object is inserted, false
            if the object already exists in the tree
 */
public boolean add(E item) {
    root = add(root, item);
    return addReturn;
}

/** Recursive add method.
    post: The data field addReturn is set true if the item is added to
          the tree, false if the item is already in the tree.
    @param localRoot The local root of the subtree
    @param item The object to be inserted
    @return The new local root that now contains the inserted item
*/
private Node<E> add(Node<E> localRoot, E item) {
    if (localRoot == null) {
        // item is not in the tree - insert it.
        addReturn = true;
        return new Node<>(item);
    } else if (item.compareTo(localRoot.data) == 0) {
        // item is equal to localRoot.data
        addReturn = false;
        return localRoot;
    } else if (item.compareTo(localRoot.data) < 0) {
        // item is less than localRoot.data
        localRoot.left = add(localRoot.left, item);
        return localRoot;
    } else {
        // item is greater than localRoot.data
        localRoot.right = add(localRoot.right, item);
        return localRoot;
    }
}
```

PROGRAM STYLE

Comment on Insertion Algorithm and add Methods

Note that as we return along the search path, the statement

```
localRoot.left = add(localRoot.left, item);
```

or

```
localRoot.right = add(localRoot.right, item);
```

resets each local root to reference the modified tree below it. You may wonder whether this is necessary. The answer is "No." In fact, it is only necessary to reset the reference from the parent of the new node to the new node; all references above the parent remain the same. We can modify the insertion algorithm to do this by checking for a leaf node before making the recursive call to `add`:

5.1. **`else if`** the item is less than `root.data`
5.2. **`if`** the local root is a leaf node.
5.3. Reset the left subtree to reference a new node with the item as its data.
`else`
5.4. Recursively insert the item in the left subtree.

A similar change should be made for the case where `item` is greater than the local root's data. You would also have to modify the starter `add` method to check for an empty tree and insert the new item in the root node if the tree is empty instead of calling the recursive `add` method.

One reason we did not write the algorithm this way is that we want to be able to adjust the tree if the insertion makes it unbalanced. This involves resetting one or more branches above the insertion point. We discuss how this is done in Chapter 9.

PROGRAM STYLE

Multiple Calls to compareTo

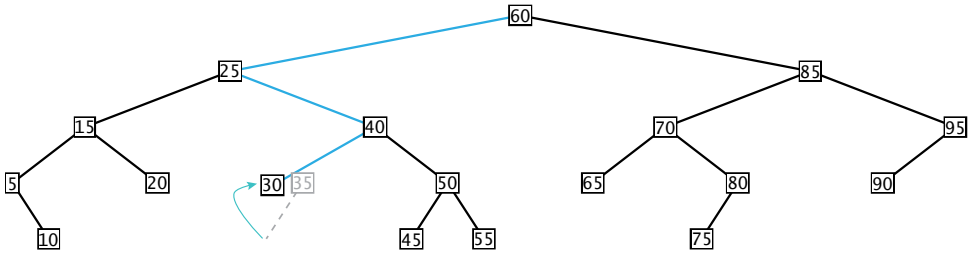
Method `add` has two calls to method `compareTo`. We wrote it this way so that the code mirrors the algorithm. However, it would be more efficient to call `compareTo` once and save the result in a local variable as we did for method `find`. Depending on the number and type of data fields being compared, the extra call to method `compareTo` could be costly.

Removal from a Binary Search Tree

Removal also follows the search algorithm except that when the item is found, it is removed. If the item is a leaf node, then its parent's reference to it is set to **`null`**, thereby removing the leaf node. If the item has only a left or right child, then the grandparent references the remaining child instead of the child's parent (the node we want to remove).

EXAMPLE 6.9 If we remove *35* from Figure 6.13, we can replace it with *30*. This is accomplished by changing the left child reference in *40* (the grandparent) to reference *30* (see Figure 6.17). The search path we follow is shown in color in Figures 6.17 through 6.19.

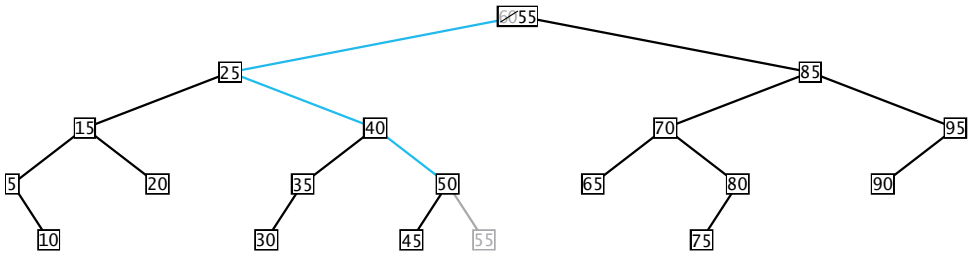
FIGURE 6.17
Removing *35*



A complication arises when the item we wish to remove has two children. In this case, we need to find a replacement parent for the children. Remember that the parent must be larger than all of the data fields in the left subtree and smaller than all of the data fields in the right subtree. If we take the largest item in the left subtree and promote it to be the parent, then all of the remaining items in the left subtree will be smaller. This item is also less than the items in the right subtree. This item is also known as the *inorder predecessor* of the item being removed. (We could use the inorder successor instead; this is discussed in the exercises.)

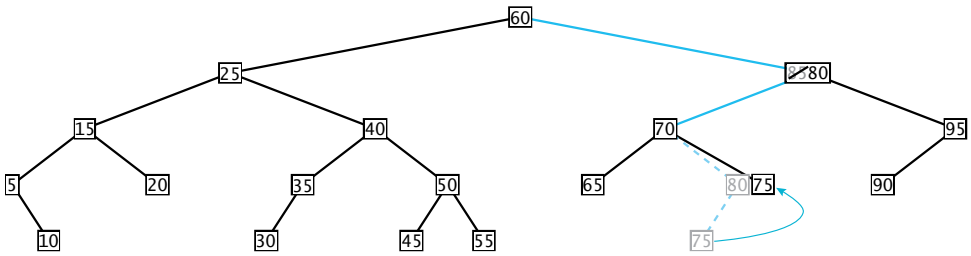
EXAMPLE 6.10 If we remove *60* from Figure 6.13, we look in the left subtree for the largest item, *55*. We then replace *60* with *55* and remove the node containing *55* (see Figure 6.18).

FIGURE 6.18
Removing *60*



EXAMPLE 6.11 If we want to remove *85* from the tree in Figure 6.13, we would terminate the search for the inorder predecessor at *80*. If we now look at *80*, we see that it does not have a right child, but it does have a left child. We would then replace *85* with its inorder predecessor, *80*, and replace the reference to *80* in *70* with a reference to *75* (the left subtree of the node that contained *80)*. See Figure 6.19.

FIGURE 6.19
Removing *85*



Recursive Algorithm for Removal from a Binary Search Tree

1. **if** the root is null
2. The item is not in tree—return **null**.
3. Compare the item to the data at the local root.
4. **if** the item is less than the data at the local root
5. Return the result of deleting from the left subtree.
6. **else if** the item is greater than the local root
7. Return the result of deleting from the right subtree.
8. **else** // *The item is in the local root*
9. Store the data in the local root in deleteReturn.
10. **if** the local root has no children
11. Set the parent of the local root to reference **null**.
12. **else if** the local root has one child
13. Set the parent of the local root to reference that child.
14. **else** // *Find the inorder predecessor*
15. **if** the left child has no right child it is the inorder predecessor
16. Set the parent of the local root to reference the left child.
17. **else**
18. Find the rightmost node in the right subtree of the left child.
19. Copy its data into the local root's data and remove it by setting its parent to reference its left child.

Implementing the delete Methods

Listing 6.6 shows both the starter and the recursive delete methods. As with the add method, the recursive delete method returns a reference to a modified tree that, in this case, no longer contains the item. The public starter method is expected to return the item removed. Thus, the recursive method saves this value in the data field deleteReturn before removing it from the tree. The starter method then returns this value.

LISTING 6.6
BinarySearchTree delete Methods

```
/** Starter method delete.
    post: The object is not in the tree.
    @param target The object to be deleted
    @return The object deleted from the tree
            or null if the object was not in the tree
    @throws ClassCastException if target does not implement
            Comparable
*/
public E delete(E target) {
    root = delete(root, target);
    return deleteReturn;
}

/** Recursive delete method.
    post: The item is not in the tree;
          deleteReturn is equal to the deleted item
          as it was stored in the tree or null
          if the item was not found.
```

```
    @param localRoot The root of the current subtree
    @param item The item to be deleted
    @return The modified local root that does not contain
            the item
*/
private Node<E> delete(Node<E> localRoot, E item) {
    if (localRoot == null) {
        // item is not in the tree.
        deleteReturn = null;
        return localRoot;
    }

    // Search for item to delete.
    int compResult = item.compareTo(localRoot.data);
    if (compResult < 0) {
        // item is smaller than localRoot.data.
        localRoot.left = delete(localRoot.left, item);
        return localRoot;
    } else if (compResult > 0) {
        // item is larger than localRoot.data.
        localRoot.right = delete(localRoot.right, item);
        return localRoot;
    } else {
        // item is at local root.
        deleteReturn = localRoot.data;
        if (localRoot.left == null) {
            // If there is no left child, return right child
            // which can also be null.
            return localRoot.right;
        } else if (localRoot.right == null) {
            // If there is no right child, return left child.
            return localRoot.left;
        } else {
            // Node being deleted has 2 children, replace the data
            // with inorder predecessor.
            if (localRoot.left.right == null) {
                // The left child has no right child.
                // Replace the data with the data in the
                // left child.
                localRoot.data = localRoot.left.data;
                // Replace the left child with its left child.
                localRoot.left = localRoot.left.left;
                return localRoot;
            } else {
                // Search for the inorder predecessor and
                // replace deleted node's data with its inorder predecessor.
                localRoot.data = findLargestChild(localRoot.left);
                return localRoot;
            }
        }
    }
}
```

For the recursive method, the two stopping cases are an empty tree and a tree whose root contains the item being removed. We first test to see whether the tree is empty (local root is **`null`**). If so, then the item sought is not in the tree. The `deleteReturn` data field is set to **`null`**, and the local root is returned to the caller.

Next, `localRoot.data` is compared to the item to be deleted. If the item to be deleted is less than `localRoot.data`, it must be in the left subtree if it is in the tree at all, so we set `localRoot.left` to the value returned by recursively calling this method.

```
localRoot.left = delete(localRoot.left, item);
```

If the item to be deleted is greater than `localRoot.data`, the statement

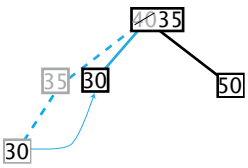
```
localRoot.right = delete(localRoot.right, item);
```

affects the right subtree of `localRoot` in a similar way.

If `localRoot.data` is the item to be deleted, we have reached the second stopping case, which begins with the lines

```
} else {
    // item is at local root.
    deleteReturn = localRoot.data;
. . .
```

The value of `localRoot.data` is saved in `deleteReturn`. If the node to be deleted has one child (or zero children), we return a reference to the only child (or **`null`**), so the parent of the deleted node will reference its only grandchild (or **`null`**).



If the node to be deleted (*40* in the figure at left) has two children, we need to find the replacement for this node. If its left child has no right subtree, the left child (*35*) is the inorder predecessor. The first statement below

```
localRoot.data = localRoot.left.data;
// Replace the left child with its left child.
localRoot.left = localRoot.left.left;
```

copies the left child's data into the local node's data (*35* to *40*); the second resets the local node's left branch to reference its left child's left subtree (*30*).

If the left child of the node to be deleted has a right subtree, the statement

```
localRoot.data = findLargestChild(localRoot.left);
```

calls `findLargestChild` to find the largest child (the inorder predecessor of the local root) and to remove it. The largest child's data is referenced by `localRoot.data`, thereby replacing its original contents. This is illustrated in Figure 6.19. The node to be deleted, *85*, has as its left child *70*. *Node 70 has a right child 80, which is its largest child and the inorder predecessor of 85*. Therefore, *80* becomes referenced by `localRoot.data` (replacing *85*) and *75* (originally the left child of *80*) becomes the new right child of *70*.

Method `findLargestChild`

Method `findLargestChild` (see Listing 6.7) takes the parent of a node as its argument. It then follows the chain of rightmost children until it finds a node whose right child does not itself have a right child. This is done via tail recursion.

When a parent node is found whose right child has no right child, the right child is the inorder predecessor of the node being deleted, so the data value from the right child is saved.

```
E returnValue = parent.right.data;
parent.right = parent.right.left;
```

The right child is then removed from the tree by replacing it with its left child (if any).

LISTING 6.7
`BinarySearchTree findLargestChild` Method

```
/** Find the node that is the
    inorder predecessor and replace it
    with its left child (if any).
    post: The inorder predecessor is removed from the tree.
    @param parent The parent of possible inorder predecessor (ip)
    @return The data in the ip
 */
```

```
private E findLargestChild(Node<E> parent) {
    // If the right child has no right child, it is
    // the inorder predecessor.
    if (parent.right.right == null) {
        E returnValue = parent.right.data;
        parent.right = parent.right.left;
        return returnValue;
    } else {
        return findLargestChild(parent.right);
    }
}
```

Testing a Binary Search Tree

To test a binary search tree, you need to verify that an inorder traversal will display the tree contents in ascending order after a series of insertions (to build the tree) and deletions are performed. You need to write a `toString` method for a `BinarySearchTree` that returns the `String` built from an inorder traversal (see Programming Exercise 3).

CASE STUDY Writing an Index for a Term Paper

Problem You would like to write an index for a term paper. The index should show each word in the paper followed by the line number on which it occurred. The words should be displayed in alphabetical order. If a word occurs on multiple lines, the line numbers should be listed in ascending order. For example, the lines

```
a, 5
a, 10
ace, 27
are, 3
are, 5
```

show that the word *a* occurred on lines 5 and 10, *ace* on line 27, and *are* on lines 3 and 5.

Analysis A binary search tree is an ideal data structure to use for storing the index entries. We can store each word and its line number as a string in a tree node. For example, the two occurrences of the word *a* on lines 5 and 10 could be stored as the strings `"a, 005"` and `"a, 010"`. Each word will be stored in lowercase to ensure that it appears in its proper position in the index. The leading zeros are necessary so that the string `"a, 005"` is considered less than the string `"a, 010"`. If the leading zeros were removed, this would not be the case (`"a, 5"` is greater than `"a, 10"`). After all the strings are stored in the search tree, we can display them in ascending order by performing an inorder traversal. Storing each word in a search tree is an $O(\log n)$ process, where n is the number of words currently in the tree. Storing each word in an ordered list would be an $O(n)$ process.

Design We can represent the index as an instance of the `BinarySearchTree` class just discussed or as an instance of a binary search tree provided in the Java API. The Java API provides a class `TreeSet<E>` (discussed further in Section 7.1) that uses a binary search tree as its basis. Class `TreeSet<E>` provides three of the methods in interface `SearchTree`: insertion (`add`), search (`boolean contains`), and removal (`boolean remove`). It also provides an iterator that enables inorder access to the elements of a tree. Because we are only doing tree insertion and inorder access, we will use class `TreeSet<E>`.

We will write a class `IndexGenerator` (see Table 6.5) with a `TreeSet<String>` data field. Method `buildIndex` will read each word from a data file and store it in the search tree. Method `showIndex` will display the index.

TABLE 6.5
Data Fields and Methods of Class `IndexGenerator`

Data Field	Attribute
`private TreeSet<String> index`	The search tree used to store the index
`private static final String PATTERN`	Pattern for extracting words from a line. A word is a string of one or more letters or numbers or characters
Method	**Behavior**
`public void buildIndex(Scanner scan)`	Reads each word from the file scanned by `scan` and stores it in tree `index`
`public void showIndex()`	Performs an inorder traversal of tree `index`

Implementation

Listing 6.8 shows class `IndexGenerator`. In method `buildIndex`, the repetition condition for the outer **`while`** loop calls method `hasNextLine`, which scans the next data line into a buffer associated with `Scanner scan` or returns `null` (causing loop exit) if all lines were scanned. If the next line is scanned, the repetition condition for the inner **`while`** loop below

```
while ((token = scan.findInLine(PATTERN)) != null) {
    token = token.toLowerCase();
    index.add(String.format("%s, %3d", token, lineNum));
}
```

calls `Scanner` method `findInLine` to extract a token from the buffer (a sequence of letters, digits, and the apostrophe character). Next, it inserts in `index` a string consisting of the next token in lowercase followed by a comma, a space, and the current line number formatted with leading spaces so that it occupies a total of three columns. This format is prescribed by the first argument `"%s, %3d"` passed to method `String.format` (see Appendix A, Section A.5). The inner loop repeats until `findInLine` returns **`null`**, at which point the inner loop is exited, the buffer is emptied by the statement

```
scan.nextLine(); // Clear the scan buffer
```

and the outer loop is repeated.

LISTING 6.8
Class `IndexGenerator.java`

```
import java.io.*;
import java.util.*;

/** Class to build an index. */
public class IndexGenerator {

    // Data Fields
    /** Tree for storing the index. */
    private final TreeSet<String> index;

    /** Pattern for extracting words from a line. A word is a string of
        one or more letters or numbers or ' characters */
    private static final String PATTERN = "[\\p{L}\\p{N}']+";

    // Methods
    public IndexGenerator() {
        index = new TreeSet<>();
    }
```

```
    /** Reads each word in a data file and stores it in an index
        along with its line number.
        post: Lowercase form of each word with its line
              number is stored in the index.
        @param scan A Scanner object
     */
    public void buildIndex(Scanner scan) {
        int lineNum = 0; // line number

        // Keep reading lines until done.
        while (scan.hasNextLine()) {
            lineNum++;

           // Extract each token and store it in index.
           String token;
           while ((token = scan.findInLine(PATTERN)) != null) {
               token = token.toLowerCase();
               index.add(String.format("%s, %3d", token, lineNum));
           }
           scan.nextLine(); // Clear the scan buffer
        }
    }

    /** Displays the index, one word per line. */
    public void showIndex() {
        index.forEach(next -> System.out.println(next));
    }
}
```

Method `showIndex` at the end of Listing 6.8 uses the default method `forEach` to display each line of the index. We describe `forEach` in the next syntax box. Without `forEach`, we could use the enhanced for loop below with an iterator.

```
public void showIndex() {
    // Use an iterator to access and display tree data.
    for (String next : index) {
         System.out.println(next);
    }
}
```

Testing To test class `IndexGenerator`, write a main method that declares new `Scanner` and `IndexGenerator<String>` objects. The `Scanner` can reference any text file stored on your computer. Make sure that duplicate words are handled properly (including duplicates on the same line), that words at the end of each line are stored in the index, that empty lines are processed correctly, and that the last line of the document is also part of the index.

SYNTAX Using the forEach Method

FORM:

```
Iterable.forEach(lambda expression);
```

EXAMPLE:

```
index.forEach(next -> System.out.println(next));
```

INTERPRETATION:

The `Iterable` interface defines a default method `forEach`. A *default method* enables you to add new functionality to an interface while still retaining compatibility with earlier implementations that did not provide this method. The `forEach` method applies a method (represented by *lambda expression)* to each item of an `Iterable` object. Since the `Set` interface (discussed in Chapter 7) extends the `Iterable` interface and `TreeSet` implements `Set`, we can use the `forEach` method on the index as shown in the example above.

EXERCISES FOR SECTION 6.5

SELF-CHECK

1. Show the tree that would be formed for the following data items. Exchange the first and last items in each list and rebuild the tree that would be formed if the items were inserted in the new order.
   a. happy, depressed, manic, sad, ecstatic
   b. 45, 30, 15, 50, 60, 20, 25, 90
2. Explain how the tree shown in Figure 6.13 would be changed if you inserted *81*. If you inserted *17*? Does either of these insertions change the height of the tree?
3. Show or explain the effect of removing the nodes *45*, *25* from the tree in Figure 6.13.
4. In Exercise 3 above, a replacement value must be chosen for the node *25* because it has two children. What is the relationship between the replacement value and *25?* What other value in the tree could also be used as a replacement for *25*? What is the relationship between that word and *25*?
5. The algorithm for deleting a node does not explicitly test for the situation where the node being deleted has no children. Explain why this is not necessary.
6. In Step 19 of the algorithm for deleting a node, when we replace the reference to a node that we are removing with a reference to its left child, why is it not a concern that we might lose the right subtree of the node that we are removing?

PROGRAMMING

1. Write methods `contains` and `remove` for the `BinarySearchTree` class. Use methods `find` and `delete` to do the work.
2. Self-Check Exercise 4 indicates that two items can be used to replace a data item in a binary search tree. Rewrite method `delete` so that it retrieves the leftmost element in the right subtree instead. You will also need to provide a method `findSmallestChild`.

3. Write a main method to test a binary search tree. Write a `toString` method that returns the tree contents in ascending order (using an inorder traversal) with newline characters separating the tree elements.
4. Write a main method for the index generator that declares new `Scanner` and `IndexGenerator` objects. The `Scanner` can reference any text file stored on your computer.

6.6 Heaps and Priority Queues

In this section, we discuss a binary tree that is ordered but in a different way from a binary search tree. At each level of a heap, the value in a node is less than all values in its two subtrees. Figure 6.20 shows an example of a heap. Observe that 6 is the smallest value. Observe that each parent is smaller than its children and that each parent has two children, with the exception of node 39 at level 3 and the leaves. Furthermore, with the exception of 66, all leaves are at the lowest level. Also, 39 is the next-to-last node at level 3, and 66 is the last (rightmost) node at level 3.

More formally, a heap is a complete binary tree with the following properties:

- The value in the root is the smallest item in the tree.
- Every subtree is a heap.

Inserting an Item into a Heap

We use the following algorithm for inserting an item into a heap. Our approach is to place each item initially in the bottom row of the heap and then move it up until it reaches the position where it belongs.

Algorithm for Inserting in a Heap

1. Insert the new item in the next position at the bottom of the heap.
2. **`while`** new item is not at the root and new item is smaller than its parent
3. Swap the new item with its parent, moving the new item up the heap.

New items are added to the last row (level) of a heap. If a new item is larger than or equal to its parent, nothing more need be done. If we insert 89 in the heap in Figure 6.20, 89 would become the right child of 39 and we are done. However, if the new item is smaller than its parent, the new item and its parent are swapped. This is repeated up the tree until the new item is in a position where it is no longer smaller than its parent. For example, let's add 8 to the heap shown in Figure 6.21. Since 8 is smaller than 66, these values are swapped as shown in Figure 6.22. Also, 8 is smaller than 29, so these values are swapped resulting in the updated heap shown in Figure 6.23. But 8 is greater than 6, so we are done.

FIGURE 6.20
Example of a Heap

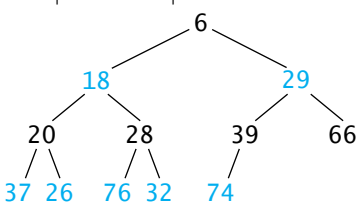


FIGURE 6.21
Inserting 8 into a Heap

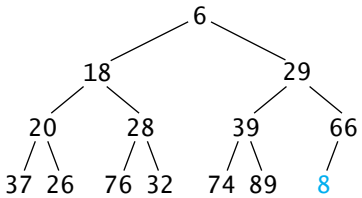


FIGURE 6.22
Swapping 8 and 66

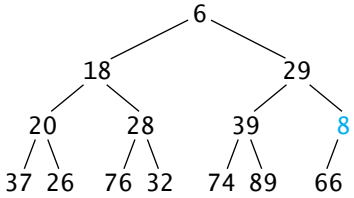
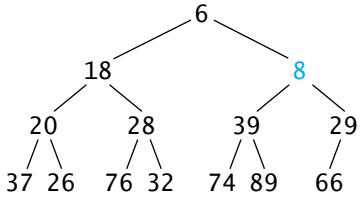


FIGURE 6.23
Swapping 8 and 29



Removing an Item from a Heap

Removal from a heap is always from the top. The top item is first replaced with the last item in the heap (at the lower right-hand position) so that the heap remains a complete tree. If we used any other value, there would be a "hole" in the tree where that value used to be. Then the new item at the top is moved down the heap until it is in its proper position.

Algorithm for Removal from a Heap

1. Remove the item in the root node by replacing it with the last item in the heap (LIH).
2. `while` item LIH has children, and item LIH is larger than either of its children
3. Swap item LIH with its smaller child, moving LIH down the heap.

As an example, if we remove 6 from the heap shown in Figure 6.23, 66 replaces it as shown in Figure 6.24. Since 66 is larger than both of its children, it is swapped with the smaller of the two, 8, as shown in Figure 6.25. The result is still not a heap because 66 is larger than both its children. Swapping 66 with its smaller child, 29, restores the heap as shown in Figure 6.26.

Implementing a Heap

Because a heap is a complete binary tree, we can implement it efficiently using an array (or `ArrayList`) instead of a linked data structure. We can use the first element (subscript 0) for storing a reference to the root data. We can use the next two elements (subscripts 1 and 2) for storing the two children of the root. We can use elements with subscripts 3, 4, 5, and 6 for storing the four children of these two nodes, and so on. Therefore, we can view a heap as a sequence of rows; each row is twice as long as the previous row. The first row (the root) has one item, the second row two, the third four, and so on. All of the rows are full except for the last one (see Figure 6.27).

FIGURE 6.24
After Removal of 6

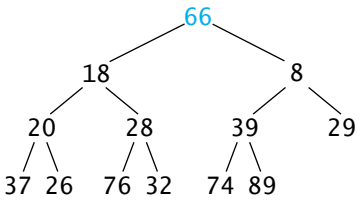


FIGURE 6.25
Swapping 66 and 8

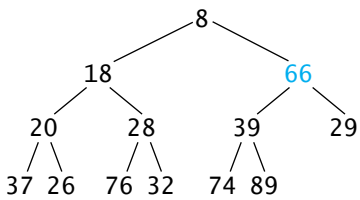


FIGURE 6.26
Swapping 66 and 29

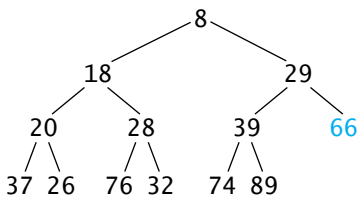
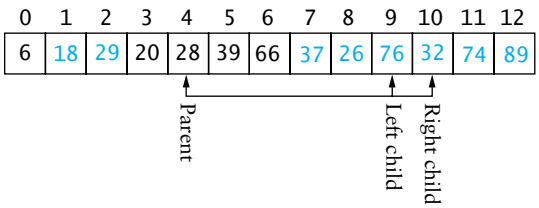
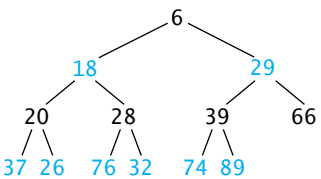


FIGURE 6.27
Internal Representation of the Heap





Observe that the root, 6, is at position 0. The root's two children, 18 and 29, are at positions 1 and 2. For a node at position p, the left child is at $2p+1$ and the right child is at $2p+2$. A node at position c can find its parent at $(c-1)/2$. Thus, as shown in Figure 6.27, children of 28 (at position 4) are at positions 9 and 10.

Insertion into a Heap Implemented as an `ArrayList`

We will use an `ArrayList` for storing our heap because it is easier to expand and contract than an array. Figure 6.28 shows the heap after inserting 8 into position 13. This corresponds to inserting the new value into the lower right position as shown in the figure, right. Now we need to move 8 up the heap, by comparing it to the values stored in its ancestor nodes. The parent (66) is in position 6 (13 minus 1 is 12, divided by 2 is 6). Since 66 is larger than 8, we need to swap as shown in Figure 6.29.

Now the child is at position 6 and the parent is at position 2 (6 minus 1 is 5, divided by 2 is 2). Since the parent, 29, is larger than the child, 8, we must swap again as shown in Figure 6.30.

The child is now at position 2 and the parent is at position 0. Since the parent is smaller than the child, the heap property is restored. In the heap insertion and removal algorithms that follow, we will use `table` to reference the `ArrayList` that stores the heap. We will use `table[index]` to represent the element at position `index` of `table`. In the actual code, a subscript cannot be used with an `ArrayList`.

Algorithm for Insertion into a Heap Implemented as an ArrayList

1. Insert the new element at the end of the `ArrayList` and set child to `table.size()` – 1.
2. Set `parent` to (`child` – 1)/2.
3. **`while`** (`parent` >= 0 and `table[parent]` > `table[child]`)
4. Swap `table[parent]` and `table[child]`.
5. Set `child` equal to `parent`.
6. Set `parent` equal to (`child` – 1)/2.

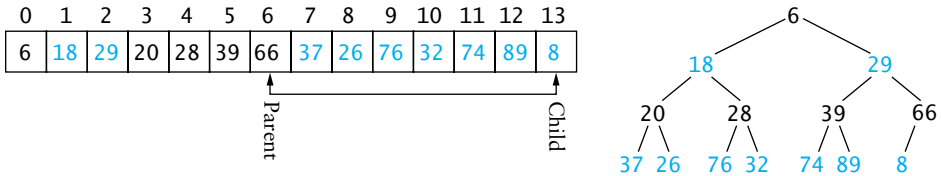


FIGURE 6.28 Internal Representation of Heap after Insertion

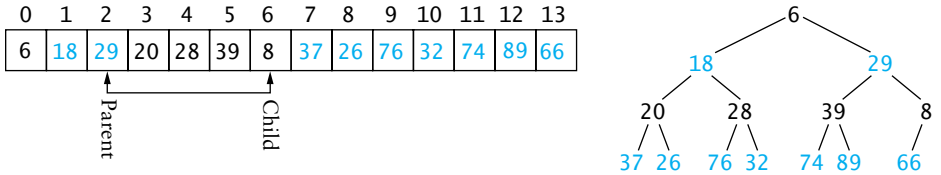


FIGURE 6.29 Internal Representation of Heap after First Swap

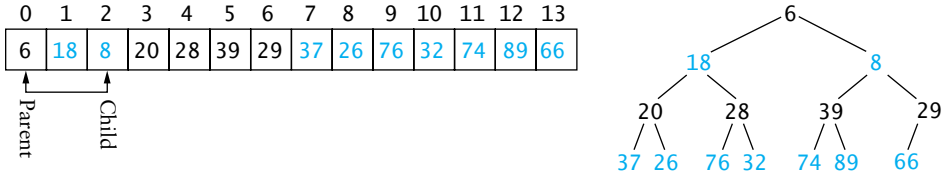


FIGURE 6.30 Internal Representation of Heap after Second Swap

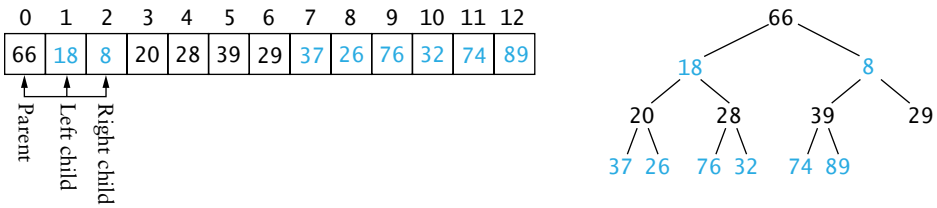


FIGURE 6.31
Internal Representation of Heap after 6 Is Removed

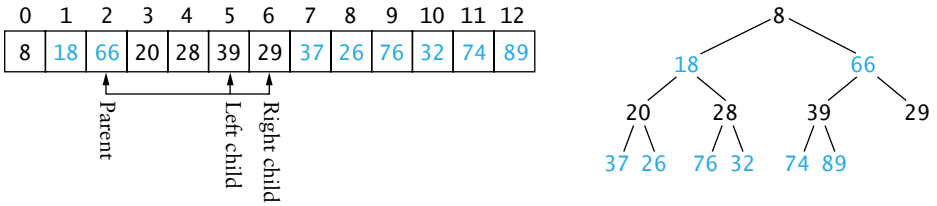


FIGURE 6.32
Internal Representation of Heap after 8 and 66 Are Swapped

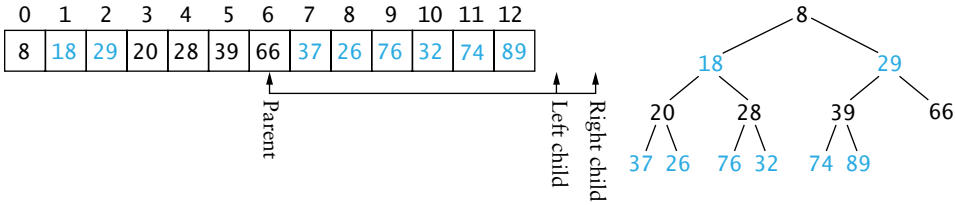


FIGURE 6.33
Internal Representation of Heap after Swap of 66 and 29

Removal from a Heap Implemented as an `ArrayList`

In removing elements from a heap, we must always remove and save the element at the top of the heap, which is the smallest element. We start with an `ArrayList` that has been organized to form a heap. To remove the first item (6), we begin by replacing the first item with the last item and then removing the last item. This is illustrated in Figure 6.31. The new value of the root (position 0) is larger than both of its children (18 in position 1 and 8 in position 2). The smaller of the two children (8 in position 2) is swapped with the parent as shown in Figure 6.32.

Next, 66 is swapped with the smaller of its two new children (29), and the heap is restored (Figure 6.33).

The algorithm for removal from a heap implemented as an `ArrayList` follows.

Algorithm for Removing an Element from a Heap Implemented as an ArrayList

1. Remove the last element (i.e., the one at `size() - 1`) and set the item at 0 to this value.
2. Set `parent` to 0.
3. **`while (true)`**
4. Set `leftChild to (2 * parent) + 1`
5. **`if`** `leftChild >= table.size()`
6. Break out of loop.
7. Set `rightChild to leftChild + 1` and `minChild` (the smaller child) to `leftChild`.
8. **`if`** `rightChild < table.size()` and `table[rightChild] < table[leftChild]`
9. Set `minChild` to `rightChild`.
10. **`if`** `table[parent] > table[minChild]`

```
11.             Swap table[parent] and table[minChild].
12.             Set parent to minChild.
            else
13.             Break out of loop.
```

The loop (Step 3) is terminated under one of two circumstances: either the item has moved down the tree so that it has no children (line 5 is true), or it is smaller than both its children (line 10 is false). In these cases, the loop terminates (line 6 or 13). This is shown in Figure 6.33. At this point, the heap property is restored, and the next smallest item can be removed from the heap.

Performance of the Heap

Method `remove` traces a path from the root to a leaf, and method `insert` traces a path from a leaf to the root. This requires at most h steps, where h is the height of the tree. The largest heap of height h is a full tree of height h. This tree has $2^h - 1$ nodes. The smallest heap of height h is a complete tree of height h, consisting of a full tree of height $h-1$, with a single node as the left child of the leftmost child at height $h-1$. Thus, this tree has $2^{(h-1)}$ nodes. Therefore, both `insert` and `remove` are $O(\log n)$, where n is the number of items in the heap.

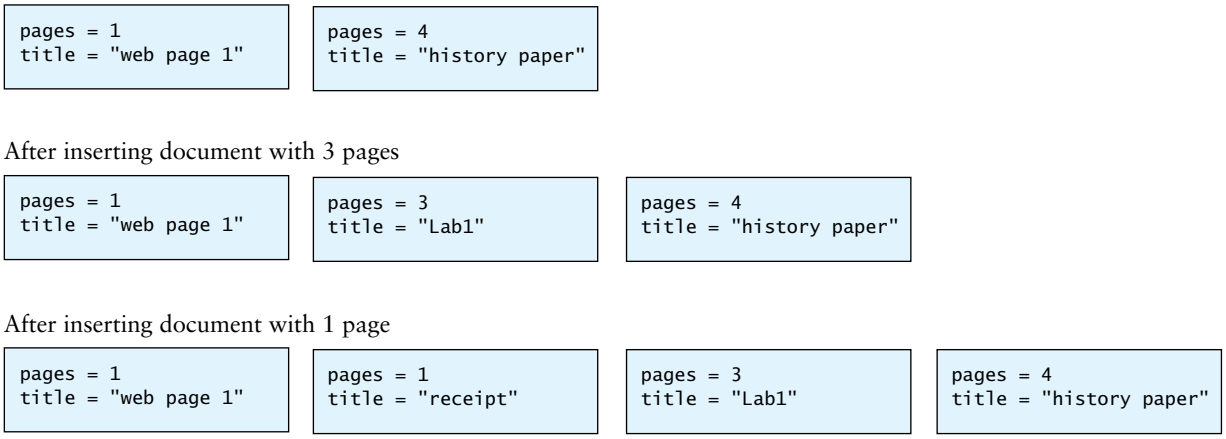
Priority Queues

In computer science, a heap is used as the basis of a very efficient algorithm for sorting arrays, called heapsort, which you will study in Chapter 8. The heap is also used to implement a special kind of queue called a priority queue. However, the heap is not very useful as an abstract data type (ADT) on its own. Consequently, we will not create a `Heap` interface or code a class that implements it. Instead we will incorporate its algorithms when we implement a priority queue class and heapsort.

Sometimes a FIFO (first-in-first-out) queue may not be the best way to implement a waiting line. In a print queue, you might want to print a short document before some longer documents that were ahead of the short document in the queue. For example, if you were waiting by the printer for a single page to print, it would be very frustrating to have to wait until several documents of 50 pages or more were printed just because they entered the queue before yours did. Therefore, a better way to implement a print queue would be to use a priority queue. A *priority queue* is a data structure in which only the highest priority item is accessible. During insertion, the position of an item in the queue is based on its priority relative to the priorities of other items in the queue. If a new item has higher priority than all items currently in the queue, it will be placed at the front of the queue and, therefore, will be removed before any of the other items inserted in the queue at an earlier time. This violates the FIFO property of an ordinary queue.

EXAMPLE 6.12 Figure 6.34 sketches a print queue that at first (top of diagram) contains two documents. We will assume that each document's priority is inversely proportional to its page count (priority is $\frac{1}{\text{page count}}$). The middle queue shows the effect of inserting a document three pages long. The bottom queue shows the effect of inserting a second one-page document. It follows the earlier document with that page length.

FIGURE 6.34
Insertion into a Priority Queue



The PriorityQueue Class

Java provides a `PriorityQueue<E>` class that implements the `Queue<E>` interface given in Chapter 4. The differences are in the specification for the `peek`, `poll`, and `remove` methods. These are defined to return the smallest item in the queue rather than the oldest item in the queue. Table 6.6 summarizes the methods of the `PriorityQueue<E>` class.

Using a Heap as the Basis of a Priority Queue

The smallest item is always removed first from a priority queue (the smallest item has the highest priority) just as it is for a heap. Because insertion into and removal from a heap is $O(\log n)$, a heap can be the basis for an efficient implementation of a priority queue. We will call our class `KWPriorityQueue` to differentiate it from class `PriorityQueue` in the `java.util` API, which also uses a heap as the basis of its implementation.

A key difference is that class `java.util.PriorityQueue` class uses an array of type `Object[]` for heap storage. We will use an `ArrayList` for storage in `KWPriorityQueue` because the size of

TABLE 6.6
Methods of the `PriorityQueue<E>` Class

Method	Behavior
`boolean offer(E item)`	Inserts an item into the queue. Returns **`true`** if successful; returns **`false`** if the item could not be inserted
`E remove()`	Removes the smallest entry and returns it if the queue is not empty. If the queue is empty, throws a `NoSuchElementException`
`E poll()`	Removes the smallest entry and returns it. If the queue is empty, returns **`null`**
`E peek()`	Returns the smallest entry without removing it. If the queue is empty, returns **`null`**
`E element()`	Returns the smallest entry without removing it. If the queue is empty, throws a `NoSuchElementException`

an ArrayList automatically adjusts as elements are inserted and removed. To insert an item into the priority queue, we first insert the item at the end of the ArrayList. Then, following the algorithm described earlier, we move this item up the heap until it is smaller than its parent.

To remove an item from the priority queue, we take the first item from the ArrayList; this is the smallest item. We then remove the last item from the ArrayList and put it into the first position of the ArrayList, overwriting the value currently there. Then, following the algorithm described earlier, we move this item down until it is smaller than its children or it has no children.

Design of KWPriorityQueue Class

The design of the KWPriorityQueue<E> class is shown in Table 6.7. The data field theData is used to store the heap.

Data field comp is a Comparator. The Comparator interface is defined as a functional interface (see Table 6.2). Its abstract method compare takes two arguments of the same type and returns an integer (–1, 0, or 1) indicating their ordering.

There are two constructors in Listing 6.9. We call the constructor without a parameter when we want to compare two objects based on their natural ordering. The lambda expression shown in the constructor implements method compare using compareTo for Comparable objects denoted as left and right.

If we want to use a different ordering for two objects, we can use the second form of the constructor that has a Comparator parameter. We should pass a lambda expression that defines the desired ordering to this parameter. We will see this later in the section when we discuss a print queue.

We have added method swap to those shown earlier in Table 6.6. The class heading and data field declarations follow.

TABLE 6.7
Design of KWPriorityQueue<E> Class

Data Field	Attribute
ArrayList<E> theData	An ArrayList to hold the data
Comparator<E> comp	An object that implements the Comparator<E> interface by providing a compare method
Method	**Behavior**
KWPriorityQueue()	Constructs a heap-based priority queue that uses the elements' natural ordering
KWPriorityQueue (Comparator<E> comp)	Constructs a heap-based priority queue that uses the compare method of Comparator comp to determine the ordering of the elements
private void swap(int i, int j)	Exchanges the object references in theData at indexes i and j

LISTING 6.9
Data Fields and Constructors for KWPriorityQueue

```
import java.util.*;
/** The KWPriorityQueue implements the Queue interface
    by building a heap in an ArrayList. The heap is structured
    so that the "smallest" item is at the top.
 */
```

```
public class KWPriorityQueue<E> extends AbstractQueue<E>
             implements Queue<E> {
    // Data Fields
    /** The ArrayList to hold the data. */
    private ArrayList<E> theData;
    /** A reference to a Comparator object. */
    Comparator<E> comp;
    // Methods
    // Constructors
    /**
     * Creates a heap-based priority queue that orders its
     * elements based on their natural ordering.
     */
    public KWPriorityQueue() {
        theData = new ArrayList<>();
        comp = (left, right) -> ((Comparable<E>) left).compareTo(right);
    }

    /**
     * Creates a heap-based priority queue that orders its
     * elements according to the specified Comparator.
     * @param comp The Comparator used to order queue elements
     */
    public KWPriorityQueue(Comparator<E> comp) {
        theData = new ArrayList<>();
        this.comp = comp;
    }
    . . .
```

The offer Method

The `offer` method (Listing 6.10) appends a new item to the `ArrayList` `theData`. It then moves this item up the heap until the `ArrayList` is restored to a heap.

LISTING 6.10

Method `KWPriorityQueue.offer`

```
/** Insert an item into the priority queue.
 * pre: theData is in heap order.
 * post: The item is in the priority queue and
 * theData is in heap order.
 * @param item The item to be inserted
 * @throws NullPointerException if the item to be
 * inserted is null.
 */
@Override
public boolean offer(E item) {
   // Add the item to the heap.
   theData.add(item);
   // child is newly inserted item.
   int child = theData.size() - 1;
   int parent = (child - 1) / 2; // Find child's parent.
   // Reheap
   while (parent >= 0 && comp.compare(theData.get(parent),
                                      theData.get(child)) > 0) {
       swap(parent, child);
       child = parent;
       parent = (child - 1) / 2;
   }
   return true;
}
```

The poll Method

The `poll` method (Listing 6.11) first saves the item at the top of the heap. If there is more than one item in the heap, the method removes the last item from the heap and places it at the top. Then it moves the item at the top down the heap until the heap property is restored. Next it returns the original top of the heap.

LISTING 6.11

Method `KWPriorityQueue.poll`

```
/** Remove an item from the priority queue
 *  pre: theData is in heap order.
 *  post: Removed smallest item, theData is in heap order.
 *  @return The item with the smallest priority value or null if empty.
 */
@Override
public E poll() {
    if (isEmpty()) {
        return null;
    }
    // Save the top of the heap.
    E result = theData.get(0);
    // If only one item then remove it.
    if (theData.size() == 1) {
        theData.remove(0);
        return result;
    }
    /* Remove the last item from the ArrayList and place it into
       the first position. */
    theData.set(0, theData.remove(theData.size() - 1));
    // The parent starts at the top.
    int parent = 0;
    while (true) {
        int leftChild = 2 * parent + 1;
        if (leftChild >= theData.size()) {
            break; // Out of heap.
        }
        int rightChild = leftChild + 1;
        int minChild = leftChild; // Assume leftChild is smaller.
        // See whether rightChild is smaller.
        if (rightChild < theData.size()
            && comp.compare(theData.get(leftChild),
                            theData.get(rightChild)) > 0) {
            minChild = rightChild;
        }
        // assert: minChild is the index of the smaller child.
        // Move smaller child up heap if necessary.
        if (comp.compare(theData.get(parent),
                         theData.get(minChild)) > 0) {
            swap(parent, minChild);
            parent = minChild;
        } else { // Heap property is restored.
            break;
        }
    }
    return result;
}
```

The Other Methods

The private method `swap` exchanges the two objects referenced by its parameters. The `iterator` and `size` methods are implemented via delegation to the corresponding `ArrayList` methods. Method `isEmpty` tests whether the result of calling method `size` is 0 and is inherited from class `AbstractCollection` (a super interface to `AbstractQueue`). Methods `peek` and `remove` (based on `poll`) must also be implemented; they are left as exercises. Methods `add` and `element` are inherited from `AbstractQueue` where they are implemented by calling methods `offer` and `peek`, respectively.

EXAMPLE 6.13 The class `PrintDocument` is used to define documents to be printed on a printer. This class implements the `Comparable` interface, but the result of its `compareTo` method is based on a lexical comparison of the names of the documents to be printed. The class also has a `getSize` method that gives the number of bytes to be transmitted to the printer and a `getTimeStamp` method that gets the time that the print job was submitted. Instead of basing the ordering on document names, we want to order the documents by a value that is a function of both size and the waiting time of a document. If we were to use either time or size alone, small documents could be delayed while big ones are printed, or big documents would never be printed. By using a priority value that is a combination, we achieve a balanced usage of the printer.

The constructor below has as its argument a lambda expression that implements `Comparator`. It compares documents `left` and `right` using the weighted sum of their size and time stamp with weighting factors `P1` and `P2`. Method `Double.compare` in this fragment compares two double values and returns a negative value, 0, or a positive value depending on whether `leftValue` is <, equal to, or > `rightValue`.

```
final double P1 = 0.8;
final double P2 = 0.2;
Queue<PrintDocument> printQueue =
    new KWPriorityQueue<>((left, right) -> {
        double leftValue = P1 * left.getSize() + P2 * left.getTimeStamp();
        double rightValue = P1 * right.getSize() + P2 * right.getTimeStamp();
        return Double.compare(leftValue, rightValue);
});
```

EXERCISES FOR SECTION 6.6

SELF-CHECK

1. Show the heap that would be used to store the words *this*, *is*, *the*, *house*, *that*, *jack*, *built*, assuming they are inserted in that sequence. Exchange the order of arrival of the first and last words and build the new heap.
2. Draw the heaps for Exercise 1 above as arrays.
3. Show the result of removing the number 18 from the heap in Figure 6.26. Show the new heap and its array representation.
4. The heaps in this chapter are called min heaps because the smallest key is at the top of the heap. A max heap is a heap in which each element has a key that is smaller than its parent, so the largest key is at the top of the heap. Build the max heap that would result from the numbers 15, 25, 10, 33, 55, 47, 82, 90, 18 arriving in that order.

5. Show the printer queue after receipt of the following documents:

time stamp	size
1100	256
1101	512
1102	64
1103	96

PROGRAMMING

1. Complete the implementation of the `KWPriorityQueue` class. Write method `swap`. Also write methods `peek`, `remove`, `isEmpty`, and `size`.

6.7 Huffman Trees

In Section 6.1, we showed the Huffman coding tree and how it can be used to decode a message. We will now implement some of the methods needed to build a tree and decode a message. We will do this using a binary tree and a `PriorityQueue` (which also uses a binary tree).

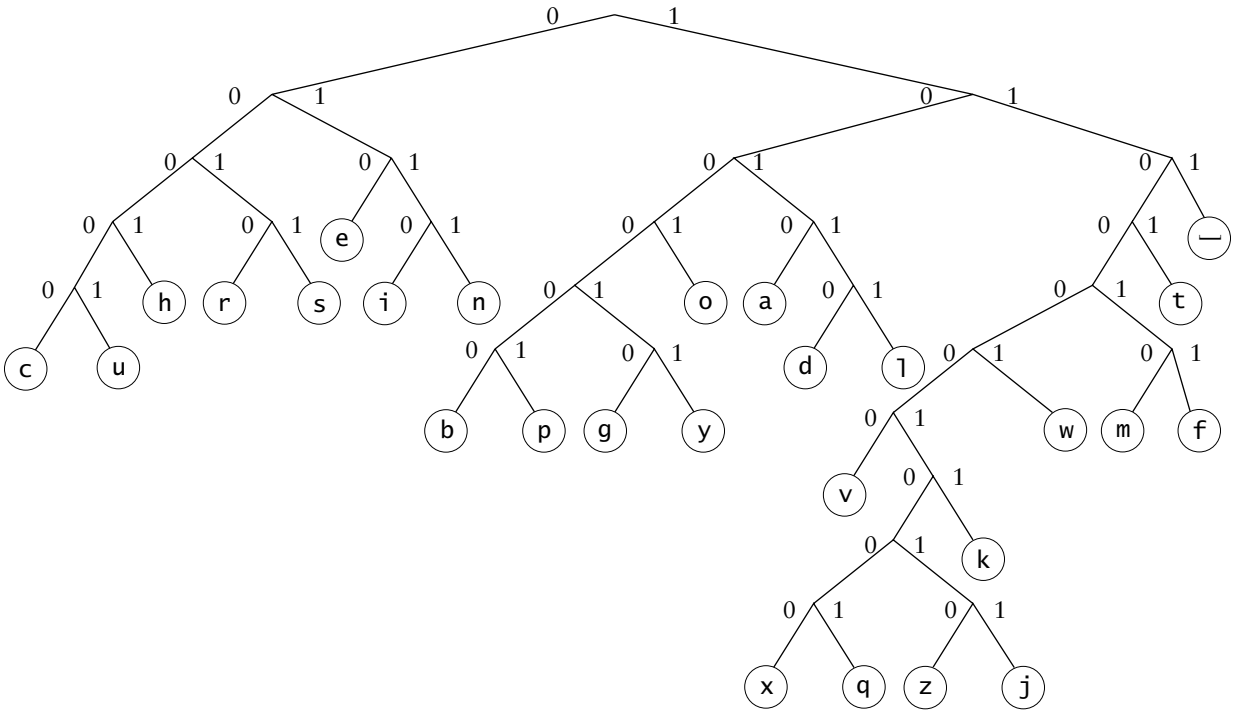
A straight binary coding of an alphabet assigns a unique binary number k to each symbol in the alphabet a_k. An example of such a coding is Unicode, which is used by Java for the **char** data type. There are 65,536 possible characters, and they are assigned a number between 0 and 65,535, which is a string of 16 binary digit ones. Therefore, the length of a message would be $16 \times n$, where n is the total number of characters in the message. For example, the message "go eagles" contains 9 characters and would require 9×16 or 144 bits. As shown in the example in Section 6.1, a Huffman coding of this message requires just 38 bits.

Table 6.8, based on data published in Donald Knuth, *The Art of Computer Programming, Vol 3: Sorting and Searching* (Addison-Wesley, 1973), p. 441, represents the relative

TABLE 6.8
Frequency of Letters in English Text

Symbol	Frequency	Symbol	Frequency	Symbol	Frequency
␣	186	h	47	g	15
e	103	d	32	p	15
t	80	l	32	b	13
a	64	u	23	v	8
o	63	c	22	k	5
i	57	f	21	j	1
n	57	m	20	q	1
s	51	w	18	x	1
r	48	y	16	z	1

FIGURE 6.35
Huffman Tree Based on Frequency of Letters in English Text



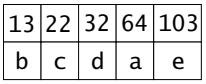
frequencies of the letters in English text and is the basis of the tree shown in Figure 6.35. The letter *e* occurs an average of 103 times every 1000 letters, or 10.3 percent of the letters are *e*s. (This is a useful table to know if you are a fan of *Wheel of Fortune*.) We can use this Huffman tree to encode and decode a file of English text. However, files may contain other symbols or may contain these symbols in different frequencies from what is found in normal English. For this reason, you may want to build a custom Huffman tree based on the contents of the file you are encoding. You would attach this tree to the encoded file so that it can be used to decode the file. We discuss how to build a Huffman tree in the next case study.

CASE STUDY Building a Custom Huffman Tree

Problem You want to build a custom Huffman tree for a particular file. Your input will consist of an array of objects such that each object contains a reference to a symbol occurring in that file and the frequency of occurrence (weight) for the symbol in that file.

Analysis Each node of a Huffman tree has storage for two data items: the weight of the node and the symbol associated with that node. All symbols will be stored at leaf nodes. For nodes that are not leaf nodes, the symbol part has no meaning. The weight of a leaf node will be the frequency of the symbol stored at that node. The weight of an interior node will be the sum of frequencies of all nodes in the subtree rooted at the interior node. For example, the interior node with leaf nodes *c* and *u* (on the left of Figure 6.35) would have a weight of 45 (22+23).

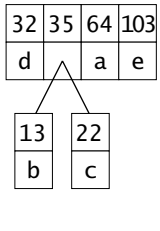
FIGURE 6.36
Priority Queue with the Symbols *a*, *b*, *c*, *d*, and *e*



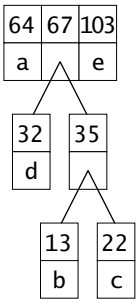
We will use a priority queue as the key data structure in constructing the Huffman tree. We will store individual symbols and subtrees of multiple symbols in order by their priority (frequency of occurrence). We want to remove symbols that occur less frequently first because they should be lower down in the Huffman tree we are constructing. We discuss how this is done next.

To build a Huffman tree, we start by inserting references to trees with just leaf nodes in a priority queue. Each leaf node will store a symbol and its weight. The queue elements will be ordered so that the leaf node with the smallest weight (lowest frequency) is removed first. Figure 6.36 shows a priority queue, containing just the symbols *a*, *b*, *c*, *d*, and *e*, that uses the weights shown in Table 6.8. The item at the front of the queue stores a reference to a tree with a root node that is a leaf node containing the symbol *b* with a weight (frequency) of 13. To represent the tree referenced by a queue element, we list the root node information for that tree.

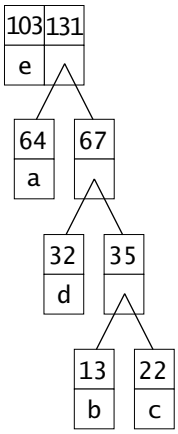
Now we start to build the Huffman tree. We build it from the bottom up. The first step is to remove the first two tree references from the priority queue and combine them to form a new tree. The weight of the root node for this tree will be the sum of the weights of its left and right subtrees. We insert the new tree back into the priority queue. The priority queue now contains references to four binary trees instead of five. The tree referenced by the second element of the queue has a combined weight of 35 (13+22) as shown on the left.



Again we remove the first two tree references and combine them. The new binary tree will have a weight of 67 in its root node. We put this tree back in the queue, and it will be referenced by the second element of the queue.



We repeat this process again. The new queue follows:



Finally, we combine the last two elements into a new tree and put a reference to it in the priority queue. Now there is only one tree in the queue, so we have finished building the Huffman tree (see Figure 6.37). Table 6.9 shows the codes for this tree.

FIGURE 6.37
Huffman Tree of *a*, *b*, *c*, *d*, and **e**

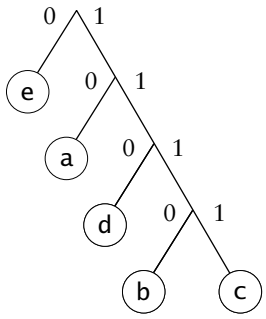


TABLE 6.9
Huffman Code for *a*, *b*, *c*, *d*, and **e**

Symbol	Code
a	10
b	1110
c	1111
d	110
e	0

The steps used to build the tree are summarized in the following algorithm.

Algorithm for Building a Huffman Tree

1. Construct a set of trees with root nodes that contain each of the individual symbols and their weights.
2. Place the set of trees into a priority queue.
3. **`while`** the priority queue has more than one item
4. Remove the two trees with the smallest weights.
5. Combine them into a new binary tree in which the weight of the tree root is the sum of the weights of its children.
6. Insert the newly created tree back into the priority queue.

Each time through the **`while`** loop, two nodes are removed from the priority queue and one is inserted. Thus, effectively one tree is removed, and the queue gets smaller with each pass through the loop.

Design

The class `HuffData` will represent the data to be stored in each node of the Huffman binary tree. For a leaf, a `HuffData` object will contain the symbol and the weight.

Our class `HuffmanTree` will have the methods and attributes listed in Table 6.10.

TABLE 6.10
Data Fields and Methods of Class `HuffmanTree`

Data Field	Attribute
`BinaryTree <HuffData> huffTree`	A reference to the Huffman tree
Method	**Behavior**
`buildTree(HuffData[] input)`	Builds the Huffman tree using the given alphabet and weights
`String decode(String message)`	Decodes a message using the generated Huffman tree
`printCode(PrintStream out)`	Outputs the resulting code

Implementation

Listing 6.12 shows the nested class `HuffData` with the constructor for a leaf and the data field declaration for class `HuffmanTree`. Method `buildTree` and the `Comparator` are discussed in the next section.

LISTING 6.12
Class `HuffmanTree` Data Field and Nested Class `HuffData`

```
import java.util.*;
import java.io.*;

/** Class to represent and build a Huffman tree. */
public class HuffmanTree {

    // Nested Classes
    /** A datum in the Huffman tree. */
    public static class HuffData {
        // Data Fields
        /** The weight or probability assigned to this HuffData. */
        private double weight;
        /** The alphabet symbol if this is a leaf. */
        private char symbol;

        public HuffData(double weight, char symbol) {
            this.weight = weight;
            this.symbol = symbol;
        }
    }

    // Data Field for HuffmanTree
    /** A reference to the completed Huffman tree. */
    private BinaryTree<HuffData> huffTree;
```

The buildTree Method

Method `buildTree` (see Listing 6.13) takes an array of `HuffData` objects as its parameter. The statement

```
Queue<BinaryTree<HuffData>> theQueue
    = new PriorityQueue<>(
            (lt, rt) -> Double.compare(lt.getData().weight,
                                       rt.getData().weight));
```

creates a new priority queue for storing `BinaryTree<HuffData>` objects using the `PriorityQueue` class in the `java.util` API. The constructor above has as its argument a lambda expression that implements `Comparator`.

The enhanced for loop loads the priority queue with trees consisting just of leaf nodes. Each leaf node contains a `HuffData` object with the weight and alphabet symbol.

The **`while`** loop builds the tree. Each time through this loop, the trees with the smallest weights are removed and referenced by `left` and `right`. The statements

```
HuffData sum = new HuffData(w1 + wr, '\u0000');
var newTree
         = new BinaryTree<HuffData>(sum, left, right);
```

combine them to form a new `BinaryTree` with a root node whose weight is the sum of the weights of its children and whose symbol is the `null` character `'\u0000'`. This new tree

is then inserted into the priority queue. The number of trees in the queue decreases by 1 each time we do this. Eventually, there will only be one tree in the queue, and that will be the completed Huffman tree. The last statement sets the variable `huffTree` to reference this tree.

LISTING 6.13

The `HuffmanTree.buildTree` Method

```
/** Builds the Huffman tree using the given alphabet and weights.
    post: huffTree contains a reference to the Huffman tree.
    @param symbols An array of HuffData objects
 */
public void buildTree(HuffData[] symbols) {
    Queue<BinaryTree<HuffData>> theQueue
        = new PriorityQueue<>(
                (lt, rt) -> Double.compare(lt.getData().weight,
                                           rt.getData().weight));
    // Load the queue with the leaves.
    for (HuffData nextSymbol : symbols) {
        var aBinaryTree =
                    new BinaryTree<HuffData>(nextSymbol, null, null);
        theQueue.offer(aBinaryTree);
    }

    // Build the tree.
    while (theQueue.size() > 1) { ,
        BinaryTree<HuffData> left = theQueue.poll();
        BinaryTree<HuffData> right = theQueue.poll();
        double wl = left.getData().weight;
        double wr = right.getDataO.weight; '
        HuffData sum = new HuffData(wl + wr, '\u0000');
        var newTree =
                 new BinaryTree<HuffData>(sum, left, right);
        theQueue.offer(newTree);
    }
    // The queue should now contain only one item.
    huffTree = theQueue.poll();
}
```

Testing

Methods `printCode` (Listing 6.14) and `decode` (Listing 6.15) can be used to test the custom Huffman tree. Method `printCode` displays the tree, so you can examine it and verify that the Huffman tree that was built is correct based on the input data.

Method `decode` will decode a message that has been encoded using the code stored in the Huffman tree and displayed by `printCode`, so you can pass it a message string that consists of binary digits only and see whether it can be transformed back to the original symbols.

We will discuss testing the Huffman tree further in the next chapter when we continue the case study.

The `HuffmanTree.printCode` Method

To display the code for each alphabet symbol, we perform a preorder traversal of the final tree. The code so far is passed as a parameter along with the current node. If the current node is a leaf, as indicated by the symbol not being **`null`**, then the code is output.

Otherwise the left and right subtrees are traversed. When we traverse the left subtree, we append a 0 to the code, and when we traverse the right subtree, we append a 1 to the code. Recall that at each level in the recursion, there is a new copy of the parameters and local variables.

LISTING 6.14
The `HuffmanTree.printCode` Method

```
/** Outputs the resulting code.
 * @param out A PrintStream for the output
 * @param code The code up to this node
 * @param tree The current node in the tree
 */
private void printCode(PrintStream out, String code,
                       BinaryTree<HuffData> tree) {
    HuffData theData = tree.getData();
    if (theData.symbol != '\u0000') {
        if (theData.symbol.equals(' ')) {
            out.println("space: " + code);
        } else {
            out.println(theData.symbol + ": " + code);
        }
    } else {
        printCode(out, code + "0", tree.getLeftSubtree());
        printCode(out, code + "1", tree.getRightSubtree());
    }
}
```

The decode Method

To illustrate the decode process, we will show a method that takes a `String` that contains a sequence of the digit characters `'0'` and `'1'` and decodes it into a message that is also a `String`. Method `decode` starts by setting `currentTree` to the Huffman tree. It then loops through the coded message one character at a time. If the character is a `'1'`, then `currentTree` is set to the right subtree; otherwise, it is set to the left subtree. If the `currentTree` is now a leaf, the symbol is appended to the result and `currentTree` is reset to the Huffman tree (see Listing 6.15). Note that this method is for testing purposes only. In actual usage, a message would be encoded as a string of bits (not digit characters) and would be decoded one bit at a time.

LISTING 6.15
Method `HuffmanTree.decode`

```
/** Method to decode a message that is input as a
 * string of digit characters '0' and '1'.
 * @param codedMessage The input message as a string of zeros and ones.
 * @return The decoded message as a String
 */
public String decode(String codedMessage) {
    var result = new StringBuilder();
    var currentTree = huffTree;
    for (int i = 0; i < codedMessage.length(); i++) {
```

```
            if (codedMessage.charAt(i) == '1') {
                currentTree = currentTree.getRightSubtree();
            } else {
                currentTree = currentTree.getLeftSubtree();
            }
            if (currentTree.isLeaf()) {
                HuffData theData = currentTree.getData();
                result.append(theData.symbol);
                currentTree = huffTree;
            }
        }
        return result.toString();
    }
```

PROGRAM STYLE

A Generic HuffmanTree Class

We chose to implement a nongeneric `HuffmanTree` class to simplify the coding. However, it may be desirable to build a Huffman tree for storing `Strings` (e.g., to encode words in a document instead of the individual letters) or for storing groups of pixels in an image file. A generic `HuffmanTree<T>` class would define a generic inner class `HuffData<T>`, where the `T` is the data type of data field `symbol`. Each parameter type `<HuffData>` in our class `HuffmanTree` would be replaced by `<HuffData<T>>`, which indicates that `T` is a type parameter for class `HuffData`.

EXERCISES FOR SECTION 6.7

SELF-CHECK

1. What is the Huffman code for the letters *a*, *j*, *k*, *l*, *s*, *t*, and *v* using Figure 6.35?
2. Trace the execution of method `printCode` for the Huffman tree in Figure 6.37.
3. Trace the execution of method `decode` for the Huffman tree in Figure 6.37 and the encoded message string `"11101011011001111"`.
4. Create the Huffman code tree for the following frequency table. Show the different states of the priority queue as the tree is built (see Figure 6.36).

Symbol	Frequency
*	50
+	30
–	25
/	10
%	5

5. What would the Huffman code look like if all symbols in the alphabet had equal frequency?

PROGRAMMING

1. Write a method `encode` for the `HuffmanTree` class that encodes a `String` of letters that is passed as its first argument. Assume that a second argument, `codes` (type `String[]`), contains the code strings (binary digits) for the symbols (space at position 0, *a* at position 1, *b* at position 2, etc.).

Chapter Review

- A tree is a recursive, nonlinear data structure that is used to represent data that is organized as a hierarchy.
- A binary tree is a collection of nodes with three components: a reference to a data object, a reference to a left subtree, and a reference to a right subtree. A binary tree object has a single data field, which references the root node of the tree.
- In a binary tree used to represent arithmetic expressions, the root node should store the operator that is evaluated last. All interior nodes store operators, and the leaf nodes store operands. An inorder traversal (traverse left subtree, visit root, traverse right subtree) of an expression tree yields an infix expression, a preorder traversal (visit root, traverse left subtree, traverse right subtree) yields a prefix expression, and a postorder traversal (traverse left subtree, traverse right subtree, visit root) yields a postfix expression.
- Java 8 lambda expressions enable a programmer to practice functional programming in Java. A lambda expression is an anonymous method with a special shorthand notation to specify its arguments and method body. A lambda interface may be assigned to an object that instantiates a functional interface. The method body of the lambda expression will implement the single abstract method of the functional interface and will execute when applied to the functional object. Java 8 provides a set of functional interfaces in library `java.util.function`. A lambda expression may also be passed to a method with a parameter that is a function object.
- A binary search tree is a tree in which the data stored in the left subtree of every node is less than the data stored in the root node, and the data stored in the right subtree of every node is greater than the data stored in the root node. The performance depends on the fullness of the tree and can range from $O(n)$ (for trees that resemble linked lists) to $O(\log n)$ if the tree is full. An inorder traversal visits the nodes in increasing order.
- A heap is a complete binary tree in which the data in each node is less than the data in both its subtrees. A heap can be implemented very effectively as an array. The children of the node at subscript p are at subscripts $2p+1$ and $2p+2$. The parent of child c is at $(c-1)/2$. The item at the top of a heap is the smallest item.
- Insertion and removal in a heap are both $O(\log n)$. For this reason, a heap can be used to efficiently implement a priority queue. A priority queue is a data structure in which the item with the highest priority (indicated by the smallest value) is removed next. The item with the highest priority is at the top of a heap and is always removed next.

- A Huffman tree is a binary tree used to store a code that facilitates file compression. The length of the bit string corresponding to a symbol in the file is inversely proportional to its frequency, so the symbol with the highest frequency of occurrence has the shortest length. In building a Huffman tree, a priority queue is used to store the symbols and trees formed so far. Each step in building the Huffman tree consists of removing two items and forming a new tree with these two items as the left and right subtrees of the new tree's root node. A reference to each new tree is inserted in the priority queue.

Java API Interfaces and Classes Introduced in This Chapter

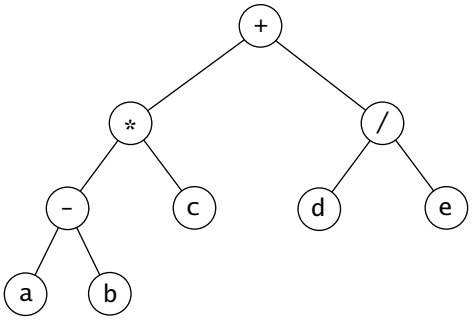
`java.util.Comparator`
`java.util.function.BiConsumer`
`java.util.function.Consumer`
`java.util.PriorityQueue`
`java.util.TreeSet`
`java.util.function.BinaryOperator`
`java.util.function.Function`

User-Defined Interfaces and Classes in This Chapter

`BinarySearchTree`
`BinaryTree`
`HuffData`
`HuffmanTree`
`IndexGenerator`
`KWPriorityQueue`
`Node`
`PrintDocument`
`PriorityQueue`
`SearchTree`

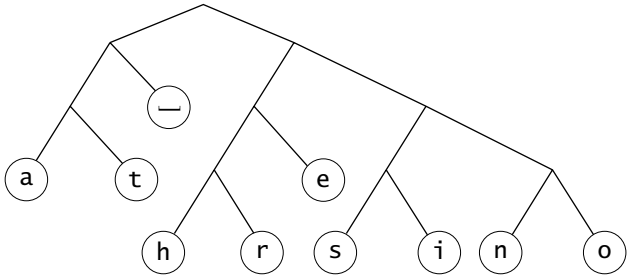
Quick-Check Exercises

1. For the following expression tree



   a. Is the tree full? _____ Is the tree complete? _____
   b. List the order in which the nodes would be visited in a preorder traversal.
   c. List the order in which the nodes would be visited in an inorder traversal.
   d. List the order in which the nodes would be visited in a postorder traversal.
2. Searching a full binary search tree is O(_____).
3. A heap is a binary tree that is a (full / complete) tree.
4. Write a lambda expression that can be used as a `Comparator` that compares two objects by weight. Assume there is method `getWeight()` that returns a `double` value.
5. Show the binary search tree that would result from inserting the items 35, 20, 30, 50, 45, 60, 18, 25 in this sequence.

6. Show the binary search tree in Exercise 5 after 35 is removed.
7. Show the heap that would result from inserting the items from Exercise 5 in the order given.
8. Draw the heap from Exercise 7 as an array.
9. Show the heap in Exercise 7 after 18 is removed.
10. In a Huffman tree, the item with the highest frequency of occurrence will have the _____ code.
11. List the code for each symbol shown in the following Huffman tree.



Review Questions

1. Draw the tree that would be formed by inserting the words in this question into a binary search tree. Use lowercase letters.
2. Show all three traversals of this tree.
3. Show the tree from Question 1 after removing *draw*, *by*, and *letters* in that order.
4. Answer Question 1 storing the words in a heap instead of a binary search tree.
5. Write a lambda expression that can be used as a predicate that returns true if an object's color is red. Assume there is a method `getColor` that returns the color as a string.
6. Given the following frequency table, construct a Huffman code tree. Show the initial priority queue and all changes in its state as the tree is constructed.

Symbol	Frequency
x	34
y	28
w	20
a	10
b	8
c	5

Programming Projects

1. Assume that a class `ExpressionTree` has a data field that is a `BinaryTree`. Write an instance method to evaluate an expression stored in a binary tree whose nodes contain either integer values (stored in `Integer` objects) or operators (stored in `Character` objects). Your method should implement the following algorithm.

Algorithm to Evaluate an Expression Tree

1. `if` the root node is an Integer object
2. Return the integer value.

3. **else if** the root node is a `Character` object
4. Let `leftVal` be the value obtained by recursively applying this algorithm to the left subtree.
5. Let `rightVal` be the value obtained by recursively applying this algorithm to the right subtree.
6. Return the value obtained by applying the operator in the root node to `leftVal` and `rightVal`.

Adapt the method `readBinaryTree` to read the expression tree.

2. Write an application to test the `HuffmanTree` class. Your application will need to read a text file and build a frequency table for the characters occurring in that file. Once that table is built, create a Huffman code tree and then a string consisting of `'0'` and `'1'` digit characters that represents the code string for that file. Read that string back in and re-create the contents of the original file.
3. Build a generic `HuffmanTree<T>` class such that the symbol type `T` is specified when the tree is created. Test this class by using it to encode the words in your favorite nursery rhyme.
4. Write `clone`, `size`, and `height` methods for the `BinaryTree` class.
5. In a breadth-first traversal of a binary tree, the nodes are visited in an order prescribed by their level. First visit the node at level 1, the root node. Then visit the nodes at level 2, in left-to-right order, and so on. You can use a queue to implement a breadth-first traversal of a binary tree.

Algorithm for Breadth-First Traversal of a Binary Tree

1. Insert the root node in the queue.
2. **while** the queue is not empty
3. Remove a node from the queue and visit it.
4. Place references to its left and right subtrees in the queue.

Code this algorithm and test it on several binary trees.

6. Define an `IndexTree` class such that each node has data fields to store a word, the count of occurrences of that word in a document file, and the line number for each occurrence. Use an `ArrayList` to store the line numbers. Use an `IndexTree` object to store an index of words appearing in a text file, and then display the index by performing an inorder traversal of this tree.
7. Extend the `BinaryTreeClass` to implement the `Iterable` interface by providing an iterator. The iterator should access the tree elements using an inorder traversal. The iterator is implemented as a nested private class. (Note: Unlike `Node`, this class should not be static.)Design hints:

   You will need a stack to hold the path from the current node back to the root. You will also need a reference to the current node (`current`) and a variable that stores the last item returned.

   To initialize `current`, the constructor should start at the root and follow the left links until a node is reached that does not have a left child. This node is the initial current node.

   The `remove` method can throw an `UnsupportedOperationException`. The `next` method should use the following algorithm:

1. Save the contents of the current node.
2. **if** the current node has a right child
3. push the current node onto the stack
4. set the current node to the right child
5. **while** the current node has a left child
6. push the current node onto the stack
7. set the current node to the left child
8. **else** the current node does not have a right child
9. **while** the stack is not empty and
   the top node of the stack's right child is equal to the current node
10. set the current node to the top of the stack and pop the stack

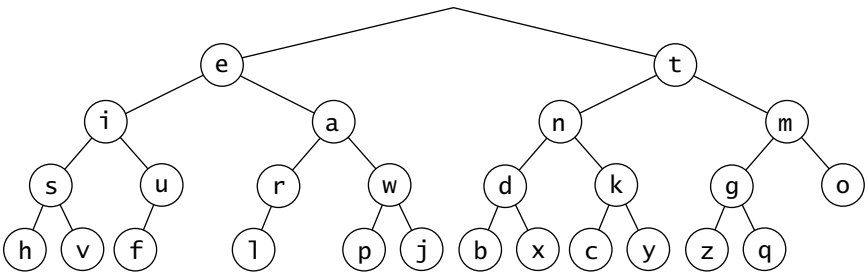
```
11.         if the stack is empty
12.             set the current node to null indicating that iteration is complete
13.         else
14.             set the current node to the top of the stack and pop the stack
15.     return the saved contents of the initial current node
```

8. The Morse code (see Table 6.11) is a common code that is used to encode messages consisting of letters and digits. Each letter consists of a series of dots and dashes; for example, the code for the letter *a* is •- and the code for the letter *b* is -•••. Store each letter of the alphabet in a node of a binary tree of level 5. The root node is at level 1 and stores no letter. The left node at level 2 stores the letter *e* (code is •), and the right node stores the letter *t* (code is -). The four nodes at level 3 store the letters with codes (••, •-, -•, --). To build the tree (see Figure 6.38), read a file in which each line consists of a letter followed by its code. The letters should be ordered by tree level. To find the position for a letter in the tree, scan the code and branch left for a dot and branch right for a dash. Encode a message by replacing each letter by its code symbol. Then decode the message using the Morse code tree. Make sure you use a delimiter symbol between coded letters.

TABLE 6.11
Morse Code for Letters

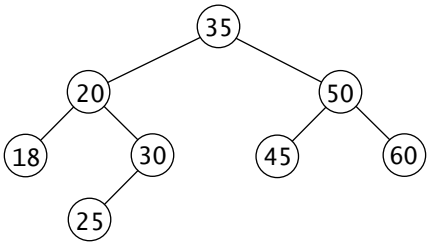
a	•–	b	–•••	c	–•–•	d	–••	e	•	f	••–•
g	––•	h	••••	i	••	j	•–––	k	–•–	l	•–••
m	––	n	–•	o	–––	p	•––•	q	––•–	r	•–•
s	•••	t	–	u	••–	v	•••–	w	•––	x	–••–
y	–•––	z	––••							

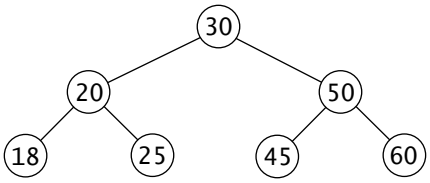
FIGURE 6.38
Morse Code Tree

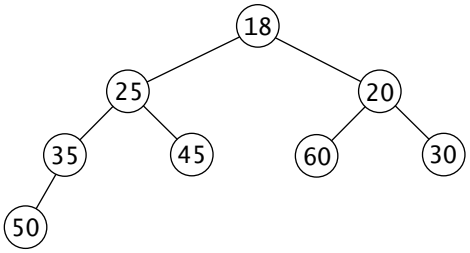


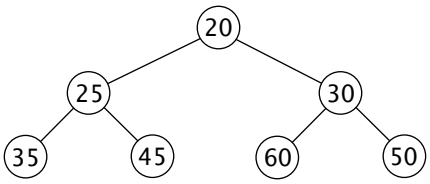
9. Create an abstract class `Heap` that has two subclasses, `MinHeap` and `MaxHeap`. Each subclass should have two constructors, one that takes no parameters and the other that takes a `Comparator` object. In the abstract class, the `compare` method should be abstract, and each subclass should define its own `compare` method to ensure that the ordering of elements corresponds to that required by the heap. For a `MinHeap`, the key in each node should be greater than the key of its parent; the ordering is reversed for a `MaxHeap`.
10. A right-threaded tree is a binary search tree in which each right link that would normally be `null` is a "thread" that links that node to its inorder successor. The thread enables nonrecursive algorithms to be written for search and inorder traversals that are more efficient than recursive ones. Implement a `RightThreadTree` class as an extension of a `BinarySearchTree`. You will also need an `RTNode` that extends the `Node` class to include a flag that indicates whether a node's right link is a real link or a thread.

Answers to Quick-Check Exercises

1. a. Not full, complete
   b. `+ * - a b c / d e`
   c. `a - b * c + d / e`
   d. `a b - c * d e / +`
2. O(log *n*)
3. A heap is a binary tree that is a *complete* tree.
4. `(o1, o2) -> Double.compare(o1.getWeightO, o2.getWeight())`
5. 

6. 

7. 

8. 18, 25, 20, 35, 45, 60, 30, 50, where 18 is at position 0 and 50 is at position 7.
9. 

10. In a Huffman tree, the item with the highest frequency of occurrence will have the *shortest* code.
11.

Symbol	Code	Symbol	Code
Space	01	n	1110
a	000	o	1111
e	101	r	1001
h	1000	s	1100
i	1101	t	001

Chapter 7

Sets and Maps

Chapter Objectives

- To understand the Java `Map` and `Set` interfaces and how to use them
- To learn about hash coding and its use to facilitate efficient search and retrieval
- To study two forms of hash tables—open addressing and chaining—and to understand their relative benefits and performance tradeoffs
- To learn how to implement both hash table forms
- To be introduced to the implementation of `Maps` and `Sets`
- To see how two earlier applications can be implemented more easily using `Map` objects for data storage
- To introduce `NavigableMaps` and `NavigableSets` and a data structure called the skip-list that can be used to implement them.

In Chapter 2, we introduced the Java Collections Framework, focusing on the `List` interface and the classes that implement it (`ArrayList` and `LinkedList`). The classes that implement the `List` interface are all indexed collections. That is, there is an index or a subscript associated with each member (element) of an object of these classes. Often an element's index reflects the relative order of its insertion in the `List` object. Searching for a particular value in a `List` object is generally an O(n) process. The exception is a binary search of a sorted object, which is an O(log n) process.

In this chapter, we consider the other part of the `Collection` hierarchy: the `Set` interface and the classes that implement it. `Set` objects are not indexed, and the order of insertion of items is not known. Their main purpose is to enable efficient search and retrieval of information. It is also possible to remove elements from these collections without moving other elements around. By contrast, if an element is removed from an `ArrayList` object, the elements that follow it are normally shifted over to fill the vacated space.

A second, related interface is the `Map`. `Map` objects provide efficient search and retrieval of entries that consist of pairs of objects. The first object in each pair is the key (a unique value), and the second object is the information associated with that key. You retrieve an object from a `Map` by specifying its key.

We also study the hash table data structure. The hash table is a very important data structure that has been used very effectively in compilers and in building dictionaries. It can be used as the underlying data structure for a `Map` or `Set` implementation. It stores objects at arbitrary locations and offers an average constant time for insertion, removal, and searching.

We will see two ways to implement a hash table and how to use it as the basis for a class that implements `Map` or `Set`. We will not show you the complete implementation of an object that implements `Map` or `Set` because we expect that you will use the ones provided by the Java API. However, we will certainly give you a head start on what you need to know to implement these interfaces.

Finally, we look at the `NavigableSet` and `NavigableMap` interfaces, which provide great flexibility in the order in which we can access elements of a `Map` or `Set`. The skip-list is a collection of sub-lists that can be used as the basis of a `NavigableMap` or `NavigableSet` and it provides O(log *n*) performance for search, insertion, and retrieval.

Sets and Maps

7.1 Sets and the Set Interface

We introduced the Java Collections Framework in Chapter 2. We covered the part of that framework that focuses on the `List` interface and its implementers. In this section, we explore the `Set` interface and its implementers.

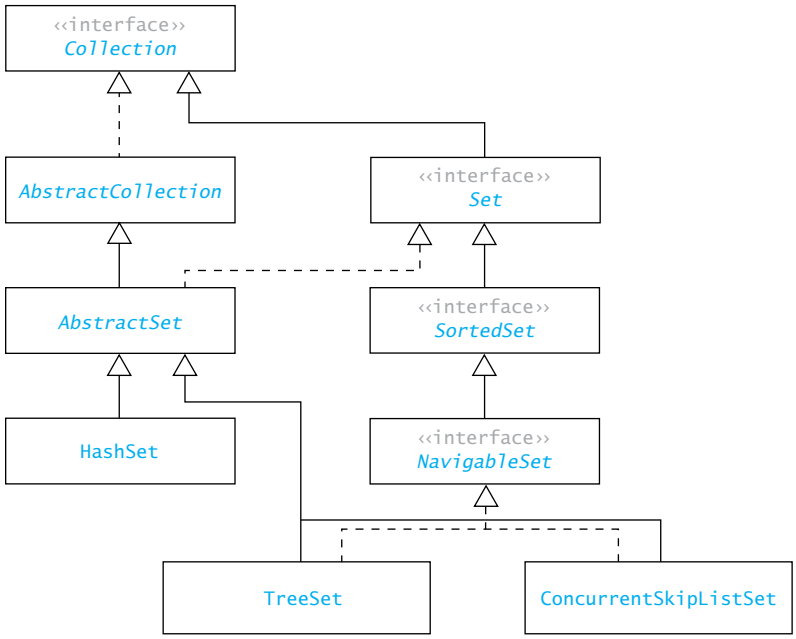
Figure 7.1 shows the part of the Collections Framework that relates to sets. It includes interfaces `Set`, `SortedSet`, and `NavigableSet`; abstract class `AbstractSet`; and actual classes `HashSet`, `TreeSet`, and `ConcurrentSkipListSet`. The `HashSet` is a set that is implemented using a hash table (discussed in Section 7.3). The `TreeSet` is implemented using a special kind of binary search tree, called the Red–Black tree (discussed in Chapter 9). The `ConcurrentSkipListSet` is implemented using a skip-list (discussed in Section 7.8). In Section 7.5, we show how to use a `TreeSet` to store an index for a term paper.

The Set Abstraction

The Java API documentation for the interface `java.util.Set` describes the `Set` as follows:

> A collection that contains no duplicate elements. More formally, sets contain no pair of elements `e1` and `e2` such that `e1.equals(e2)`, and at most one **`null`** element. As implied by its name, this interface models the mathematical *set* abstraction.

FIGURE 7.1
The Set Hierarchy



What mathematicians call a *set* can be thought of as a collection of objects. There is the additional requirement that the elements contained in the set are unique. For example, if we have the set of fruits `{"apples", "oranges", "pineapples"}` and add `"apples"` to it, we still have the same set. Also, we usually want to know whether or not a particular object is a member of the set rather than where in the set it is located. Thus, if `s` is a set, we would be interested in the expression

```
s.contains("apples")
```

which returns the value **`true`** if `"apples"` is in set `s` and **`false`** if it is not. We would not have a need to use a method such as

```
s.indexOf("apples")
```

which might return the location or position of `"apples"` in set `s`. Nor would we have a need to use the expression

```
s.get(i)
```

where `i` is the position (index) of an object in set `s`.

We assume that you are familiar with sets from a course in discrete mathematics. Just as a review, however, the operations that are performed on a mathematical set are testing for membership (method `contains`), adding elements, and removing elements. Other common operations on a mathematical set are *set union* (`A ∪ B`), *set intersection* (`A ∩ B`), and *set difference* (`A - B`). There is also a *subset operator* (`A ⊂ B`). These operations are defined as follows:

- The union of two sets `A, B` is a set whose elements belong either to `A` or `B` or to both `A` and `B`.

  Example: {1, 3, 5, 7} ∪ {2, 3, 4, 5} is {1, 2, 3, 4, 5, 7}
- The intersection of sets `A, B` is the set whose elements belong to both `A` and `B`.

  Example: {1, 3, 5, 7} ∩ {2, 3, 4, 5} is {3, 5}
- The difference of sets `A, B` is the set whose elements belong to `A` but not to `B`.

  Examples: {1, 3, 5, 7} – {2, 3, 4, 5} is {1, 7}; {2, 3, 4, 5} – {1, 3, 5, 7} is {2, 4}

- Set A is a subset of set B if every element of set A is also an element of set B.

  Example: {1, 3, 5, 7} ⊂ {1, 2, 3, 4, 5, 7} is **true**

The Set Interface and Methods

A Set has required methods for testing for set membership (contains), testing for an empty set (isEmpty), determining the set size (size), and creating an iterator over the set (iterator). It has optional methods for adding an element (add) and removing an element (remove). It provides the additional restriction on constructors that all sets they create must contain no duplicate elements. It also puts the additional restriction on the add method that a duplicate item cannot be inserted. Table 7.1 shows the commonly used methods of the Set interface. The Set interface also has methods that support the mathematical set operations. The required method containsAll tests the subset relationship. There are optional methods for set union (addAll), set intersection (retainAll), and set difference (removeAll). We show the methods that are used to implement the mathematical set operations in italics in Table 7.1.

Calling a method "optional" means that an implementer of the Set interface is not required to provide a method that performs this operation. However, a method that matches the signature must be provided in order for the implementation to compile. This method should throw an UnsupportedOperationException if it is called. This gives the class designer some flexibility. For example, if a class instance is intended to provide efficient search and retrieval of the items stored, the class designer may decide to omit the optional mathematical set operations.

TABLE 7.1
Some java.util.Set<E> Methods (with Mathematical Set Operations in Italics)

Method	Behavior
boolean add(E obj)	Adds item obj to this set if it is not already present (optional operation) and returns **true**. Returns **false** if obj is already in the set
boolean addAll(Collection<E> coll)	Adds all of the elements in collection coll to this set if they're not already present (optional operation). Returns **true** if the set is changed. Implements *set union* if coll is a Set
boolean contains(Object obj)	Returns **true** if this set contains an element that is equal to obj. Implements a test for *set membership*
boolean containsAll(Collection<E> coll)	Returns **true** if this set contains all of the elements of collection coll. If coll is a set, returns **true** if this set is a subset of coll
boolean isEmpty()	Returns **true** if this set contains no elements
Iterator<E> iterator()	Returns an iterator over the elements in this set
boolean remove(Object obj)	Removes the set element equal to obj if it is present (optional operation). Returns **true** if the object was removed
boolean removeAll(Collection<E> coll)	Removes from this set all of its elements that are contained in collection coll (optional operation). Returns **true** if this set is changed. If coll is a set, performs the *set difference* operation
boolean retainAll(Collection<E> coll)	Retains only the elements in this set that are contained in collection coll (optional operation). Returns **true** if this set is changed. If coll is a set, performs the *set intersection* operation
int size()	Returns the number of elements in this set (its cardinality)

FOR PYTHON PROGRAMMERS

The Python `Set` class is similar to the Java `HashSet` class. Both have operations for creating sets, adding and removing objects, and forming union, intersection, and difference.

EXAMPLE 7.1 Listing 7.1 contains a `main` method that creates two sets (`setA`, `setB`) and two arrays of string (`listA`, `listB`). It uses two enhanced `for` loops to add each string in an array to its corresponding set. Then it forms their union in `setA` and their intersection in `setAcopy` (a copy of the original `setA`) using the statements

```
var setAcopy = new HashSet(setA);  //copy setA
setA.addAll(setB);                  //setA is union
setAcopy.retainAll(setB);          //setAcopy is intersection
```

Running this method generates the output lines below. In the `println` statements, method `Set.toString` puts brackets around a set and commas between set elements.

```
The 2 sets are:
[Jill, Ann, Sally]
[Bill, Jill, Ann, Bob]
Items in set union are: [Bill, Jill, Ann, Sally, Bob]
Items in set intersection are: [Jill, Ann]
```

LISTING 7.1
Illustrating the Use of Sets

```
public static void main(String[] args) {
    // Create 2 empty sets and 2 arrays of strings.
    Set<String> setA = new HashSet<>();
    Set<String> setB = new HashSet<>();
    String[] listA = {"Ann", "Sally", "Jill", "Sally"};
    String[] listB = {"Bob", "Bill", "Ann", "Jill"};

    // Load sets from arrays.
    for (String s : listA) {
        setA.add(s);
    }
    for (String s : listB) {
        setB.add(s);
    }
    System.out.println("The 2 sets are: " + "\n" + setA
                       + "\n" + setB);

    // Form the set union in setA and intersection in setAcopy.
    var setAcopy = new HashSet<>(setA); // Copy setA
    setA.addAll(setB);                  // SetA is union
    setAcopy.retainAll(setB);           // SetAcopy is intersection
    System.out.println("Items in set union are: " + setA);
    System.out.println("Items in set intersection are: "
                       + setAcopy);
}
```

Using Method of to Initialize a Collection

Java 9 introduced the factory method `of` to simplify the initialization of collections. The `of` method is a static method in the `List`, `Set`, and `Map` interfaces. This method allows you to declare and initialize a set using a single statement. The two `jshell` commands below use `Set.of` to create `setB` and `List.of` to create `setA` in Listing 7.1. The sets are displayed after each `jshell` command.

```
jshell> var setB = Set.of("Bob", "Bill", "Ann", "Jill");
setB ==> [Ann, Bob, Bill, Jill]

jshell> var setA = new HashSet<>(List.of("Ann", "Sally", "Jill", "Sally"));
setA ==> [Ann, Sally, Jill]
```

Sets created using `Set. of` are *immutable*, which means they cannot be modified using `Set` operators. Therefore, attempting to apply them to `setB` would throw an `UnsupportedOperationException`. Note that the elements of `setB` are not displayed in the same order as they appear in the argument.

However, `setA` in Listing 7.1 needs to be modifiable, so we can't create it in the same way as `setB`. In the second `jshell` command above, `List.of` creates an immutable unnamed list and the `HashSet` copy constructor creates a new modifiable set whose elements correspond to the elements of the list. Another reason we cannot use `Set.of` is the duplicate element "`Sally`" would cause an `IllegalArgumentException` if passed as an argument to `Set.of`.

SYNTAX Factory method of

FORM:

`CollectionClass``.of(`*`list-of-elements`*`)`

EXAMPLE:

```
var mySet = Set.of("a", "b", "apple", "butter");
var myList = List.of(20, 35, 40, 99);
```

MEANING:

The static factory method `of` creates a new immutable object of type *`CollectionClass`*. The objects created are immutable. Attempting to modify them would throw an `UnsupportedOperationException`. For `Set.of`, the *list-of-elements* cannot contain any duplicates.

Comparison of Lists and Sets

Collections implementing the `Set` interface must contain unique elements. Unlike the `List.add` method, the `Set.add` method will return **`false`** if you attempt to insert a duplicate item.

Unlike a `List`, a `Set` does not have a `get` method. Therefore, elements cannot be accessed by index. So if `setA` is a `Set` object, the method call `setA.get(0)` would cause the syntax error `method get(int)` not found.

Although you can't reference a specific element of a `Set`, you can iterate through all its elements using an `Iterator` object. The loop below accesses each element of `Set` object `setA`.

However, the elements will be accessed in arbitrary order. This means that they will not necessarily be accessed in the order in which they were inserted.

```
// Create an iterator to setA.
Iterator<String> setAIter = setA.iterator();
while (setAIter.hasNext()) {
    String nextItem = setAIter.next();
    // Do something with nextItem
    ...
}
```

We can simplify the task of accessing each element in a `Set` using the enhanced **`for`** statement.

```
for (String nextItem : setA) {
    // Do something with nextItem
    ...
```

EXERCISES FOR SECTION 7.1

SELF-CHECK

1. Explain the effect of the following method calls.

```
Set<String> s = new HashSet<>();
s.add("hello");
s.add("bye");
s. addAll(s);
Set<String> t = new TreeSet<>();
t. add("123"); s.addAll(t);
System.out.println(s.containsAll(t));
System.out.println(t.containsAll(s));
System.out.println(s.contains("ace"));
System.out.println(s.contains("123"));
s. retainAll(t);
System.out.println(s.contains("123"));
t. retainAll(s);
System.out.println(t.contains("123"));
```

2. What is the relationship between the `Set` interface and the `Collection` interface?
3. What are the differences between the `Set` interface and the `List` interface?
4. In Example 7.1, why is `setAcopy` needed? What would happen if you used the statement

   `setAcopy = setA;`

   to define `setAcopy`?

PROGRAMMING

1. Assume you have declared three sets `a`, `b`, and `c` and that sets `a` and `b` store objects. Write statements that use methods from the `Set` interface to perform the following operations:
   a. `c = (a ∪ b)`
   b. `c = (a ∩ b)`
   c. `c = (a - b)`
   d.
```
if (a ⊂ b)
      c = a;
  else
      c = b;
```
2. Write a `toString` method for a class that implements the `Set` interface and displays the set elements in the form shown in Example 7.1.

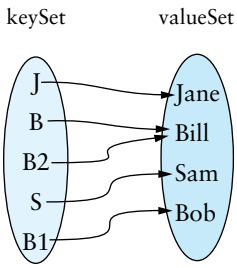
7.2 Maps and the Map Interface

The Map is related to the Set. Mathematically, a Map is a set of ordered pairs whose elements are known as the key and the value. The key is required to be unique, as are the elements of a set, but the value is not necessarily unique. For example, the following would be a map:

```
{(J, Jane), (B, Bill), (S, Sam), (B1, Bob), (B2, Bill)}
```

The keys in this example are strings consisting of one or two characters, and each value is a person's name. The keys are unique but not the values (there are two Bills). The key is based on the first letter of the person's name. The keys **B1** and **B2** are the keys for the second and third person whose name begins with the letter **B**.

FIGURE 7.2
Example of Mapping



You can think of each key as "mapping" to a particular value (hence the name *map*). For example, the key **J** maps to the value Jane. The keys **B** and **B2** map to the value Bill. You can also think of the keys as forming a set (keySet) and the values as forming a set (valueSet). Each element of keySet maps to a particular element of valueSet, as shown in Figure 7.2. In mathematical set terminology, this is a *many-to-one mapping* (i.e., more than one element of keySet may map to a particular element of valueSet). For example, both keys **B** and **B2** map to the value Bill. This is also an *onto mapping* in that all elements of valueSet have a corresponding member in keySet.

A Map can be used to enable efficient storage and retrieval of information in a table. The key is a unique identification value associated with each item stored in a table. As you will see, each key value has an easily computed numeric code value.

EXAMPLE 7.2

When information about an item is stored in a table, the information stored may consist of a unique ID (identification code, which may or may not be a number) as well as descriptive data. The unique ID would be the key, and the rest of the information would represent the value associated with that key. Some examples follow.

Type of Item	Key	Value
University student	Student ID number	Student name, address, major, grade-point average
Customer for online store	E-mail address	Customer name, address, credit card information, shopping cart
Inventory item	Part ID	Description, quantity, manufacturer, cost, price

In the above examples, the student ID number may be assigned by the university, or it may be the student's social security number. The e-mail address is a unique address for each customer, but it is not numeric. Similarly, a part ID could consist of a combination of letters and digits.

In comparing maps to indexed collections, you can think of the keys as selecting the elements of a map, just as indexes select elements in a List object. The keys for a map, however, can have arbitrary values (not restricted to 0, 1, etc., as for indexes). As you will see later, an implementation of the Map interface should have methods of the form

```
V get(Object key)
V put(K key, V value)
```

The get method retrieves the value corresponding to a specified key; the put method stores a key–value pair in a map.

The Map Hierarchy

Figure 7.3 shows part of the Map hierarchy in the Java API. Although not strictly part of the Collection hierarchy, the Map interface defines a structure that relates elements in one set to elements in another set. The first set, called the *keys*, must implement the Set interface; that is, the *keys* are unique. The second set is not strictly a Set but an arbitrary Collection known as the *values*. These are not required to be unique. The Map is a more useful structure than the Set. In fact, the Java API implements the Set using a Map.

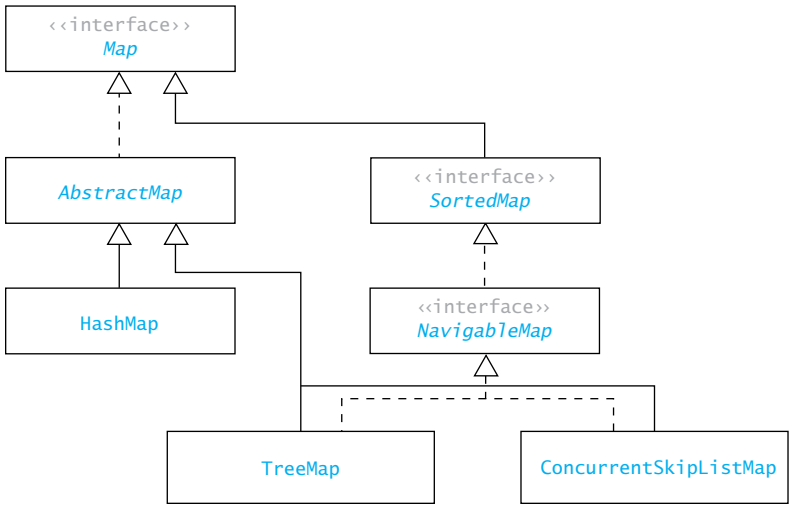


FIGURE 7.3
The Map Hierarchy

The TreeMap uses a Red–Black binary search tree (discussed in Chapter 9) as its underlying data structure, and the ConcurrentSkipListMap uses a skip-list (discussed in Section 7.8) as its underlying data structure. We will focus on the HashMap and show how to implement it later in the chapter.

The Map Interface

Methods of the Map interface (in Java API java.util) are shown in Table 7.2. The put method either inserts a new mapping or changes the value associated with an existing mapping. The get method returns the current value associated with a given key or **null** if there is none. The getOrDefault method returns the provided defaultValue instead of **null** if the key is not present. The remove method deletes an existing mapping.

The getOrDefault method is a default method, which means that the implementation of this method is defined in the interface. The code for getOrDefault follows. The test for containsKey(key) insures that the correct value (**null**) will be returned if key is present but maps to **null**.

```
default V getOrDefault(Object key, V defaultValue) {
    V value = get(key);
    if (value != null || containsKey(key)) {
        return value;
    }
    return defaultValue;
}
```

TABLE 7.2
Some `java.util.Map<K, V>` Methods

Method	Behavior
`V get(Object key)`	Returns the value associated with the specified `key`. Returns **`null`** if the `key` is not present
`boolean containsKey(Object key)`	Returns **`true`** if this map contains a mapping for the specified `key`.
`V getOrDefault(Object key, V default)`	Returns the value associated with the specified `key`. Returns `default` if the `key` is not present
`boolean isEmpty()`	Returns **`true`** if this map contains no `key-value` mappings
`V put(K key, V value)`	Associates the specified `value` with the specified `key` in this map (optional operation). Returns the previous `value` associated with the specified `key`, or `null` if there was no mapping for the `key`
`V remove(Object key)`	Removes the mapping for this `key` from this map if it is present (optional operation). Returns the previous `value` associated with the specified `key`, or **`null`** if there was no mapping for the `key`
`void forEach(BiConsumer<K, V>)`	Performs the action given by the `BiConsumer` to each entry in the map, binding the `key` to the first parameter and the value to the second
`int size()`	Returns the number of `key-value` mappings in this map

Both `put` and `remove` return the previous value (or **`null`**, if there was none) of the mapping that is changed or deleted. There are two type parameters, `K` and `V`, and they represent the data type of the key and value, respectively. We discuss method `forEach` in Example 7.12.

Creating a Map

EXAMPLE 7.3 The following statements build a `Map` object that contains the mapping shown in Figure 7.2.

```
Map<String, String> aMap = new HashMap<>();  // HashMap implements Map
aMap.put("J", "Jane");
aMap.put("B", "Bill");
aMap.put("S", "Sam");
aMap.put("B1", "Bob");
aMap.put("B2", "Bill");
```

The statement

```
System.out.println("B1 maps to " + aMap.get("B1"));
```

would display `"B1 maps to Bob"`. The statement

```
System.out.println("Bill maps to " + aMap.get("Bill"));
```

would display `"Bill maps to null"` because `"Bill"` is a value, not a key.

EXAMPLE 7.4 The factory method `Map.of` can be used to create an immutable `map`. In the `jshell` commands below, the argument for `Map.of` consists of a key followed by its value. The `maps` are displayed using `AbstractMap.toString`. The first `map` corresponds to Figure 7.2.

```
jshell> var aMap = Map.of("J","Jane","B","Bill","S","Sam","B1","Bob","B2",
"Bill");
aMap ==> {S=Sam, J=Jane, B1=Bob, B=Bill, B2=Bill}
jshell> var code = Map.of("A",1,"B",2,"C",3,"D",4,"E",5);
aMap ==> {B=2, A=1, D=4, E=5, C=3}
```

The second `jshell` command creates a `Map<String, Integer>` that maps each of the first five letters to its position in the alphabet. The compiler is able to infer the `key` and `value` types from the argument. Note that the mappings are not displayed in the order they were created.

EXAMPLE 7.5

In Section 6.5, we used a binary search tree to store an index of words occurring in a term paper. Each data element in the tree was a string consisting of a word followed by a three-digit line number.

Although this is one approach to storing an index, it would be more useful to store each word and all the line numbers for that word as a single index entry. We could do this by storing the index in a `Map` in which each word is a key and its associated value is a list of all the line numbers at which the word occurs. While building the index, each time a word is encountered, its list of line numbers would be retrieved (using the word as a key) and the most recent line number would be appended to this list (a `List<Integer>`). For example, if the word *fire* has already occurred on lines 4 and 8 and we encounter it again on line 20, the `List<Integer>` associated with *fire* would reference three `Integer` objects wrapping the numbers 4, 8, and 20.

Listing 7.2 shows method `buildIndex` (adapted from `buildIndex` in Listing 6.8). Data field `index` is a `Map` with key type `String` and value type `List<Integer>`.

```
private Map<String, List<Integer>> index;
```

The statement

```
List<Integer> lines = index.getOrDefault(token, new ArrayList<>());
```

retrieves the value (an `ArrayList<Integer>`) associated with `token` or an empty `ArrayList` if this is the first occurrence of `token`. The statements

```
lines.add(lineNum);
index.put(token, lines); // Store the list.
```

add the new line number to the `ArrayList lines` and store it back in the `Map`. In Section 7.5, we show how to display the final index.

LISTING 7.2

Method buildIndexAllLines

```
/** Reads each word in a data file and stores it in an index
    along with a list of line numbers where it occurs.
    post: Lowercase form of each word with its line
          number is stored in the index.
    @param scan A Scanner object
 */
public void buildIndex(Scanner scan) {
    int lineNum = 0; // Line number

    // Keep reading lines until done.
    while (scan.hasNextLine()) {
        lineNum++;
```

```
            // Extract each token and store it in index.
            String token;
            while ((token = scan.findInLine(PATTERN)) != null) {
                token = token.toLowerCase();
                // Get the list of line numbers for token
                List<Integer> lines = index.getOrDefault(token, new ArrayList<>());
                lines.add(lineNum);
                index.put(token, lines); // Store new list of line numbers
            }
            scan.nextline(); // Clear the scan buffer
        }
    }
```

EXERCISES FOR SECTION 7.2

SELF-CHECK

1. If you were using a `Map` to store the following lists of items, which data field would you select as the key, and why?
   a. textbook title, author, ISBN (International Standard Book Number), year, publisher
   b. player's name, uniform number, team, position
   c. computer manufacturer, model number, processor, memory, disk size
   d. department, course title, course ID, section number, days, time, room
2. For the `Map index` in Example 7.5, what key-value pairs would be stored for each token in the following data file?

```
this line is first
and line 2 is second
followed by the third line
```

3. Explain the effect of each statement in the following fragment on the index built in Self-Check Exercise 2.

```
lines = index.get("this");
lines = index.get("that");
lines = index.get("line");
lines.add(4);
index.put("is", lines);
```

PROGRAMMING

1. Write statements to create a `Map` object that will store each word occurring in a term paper along with the number of times the word occurs.
2. Write a method `buildWordCounts` (based on `buildIndex`) that builds the `Map` object described in Programming Exercise 1.

7.3 Hash Tables

Before we discuss the details of implementing the required methods of the `Set` and `Map` interfaces, we will describe a data structure, the *hash table*, that can be used as the basis for such an implementation. The goal behind the hash table is to be able to access an entry based on

its key value, not its location. In other words, we want to be able to access an element directly through its key value rather than having to determine its location first by searching for the key value in an array. (This is why the `Set` interface has method `contains(obj)` instead of `get(index)`.) Using a hash table enables us to retrieve an item in constant time (expected O(1)). We say expected O(1) rather than just O(1) because there will be some cases where the performance will be much worse than O(1) and may even be $O(n)$, but on the average, we expect that it will be O(1). Contrast this with the time required for a linear search of an array, $O(n)$, and the time to access an element in a binary search tree, $O(\log n)$.

Hash Codes and Index Calculation

The basis of hashing (and hash tables) is to transform the item's key value to an integer value (its *hash code*) that will then be transformed into a table index. Figure 7.4 illustrates this process for a table of size n. We discuss how this might be done in the next few examples.

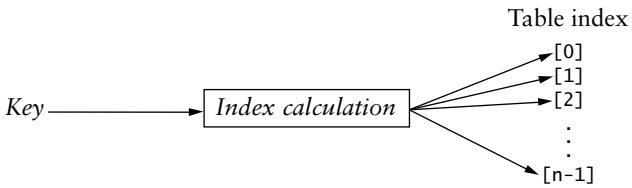


FIGURE 7.4 Index Calculation for a Key

EXAMPLE 7.6 Consider the Huffman code problem discussed in Section 6.7. To build the Huffman tree, you needed to know the number of occurrences of each character in the text being encoded. Let's assume that the text contained only the ASCII characters (the first 128 Unicode values starting with `\u0000`). We could use a table of size 128, one element for each possible character, and let the Unicode for each character be its location in the table. Using this approach, table element 65 would give us the number of occurrences of the letter A, table element 66 would give us the number of occurrences of the letter B, and so on. The hash code for each character is its Unicode value (a number), which is also its index in the table. In this case, we could calculate the table index for character `asciiChar` using the following assignment statement, where `asciiChar` represents the character we are seeking in the table:

```
int index = asciiChar;
```

EXAMPLE 7.7 Let's consider a slightly harder problem: assume that any of the Unicode characters can occur in the text, and we want to know the number of occurrences of each character. There are over 65,000 Unicode characters, however. For any file, let's assume that at most 100 different characters actually appear. So, rather than use a table with 65,536 elements, it would make sense to try to store these items in a much smaller table (say, 200 elements). If the hash code for each character is its Unicode value, we need to convert this value (between 0 and 65,535) to an array index between 0 and 199. We can calculate the array index for character `uniChar` as

```
int index = uniChar % 200
```

Because the range of Unicode values (the key range) is much larger than the `index` range, it is likely that some characters in our text will have the same index value. Because we can store only one key–value pair in a given array element, a situation known as a *collision* results. We discuss how to deal with collisions shortly.

Methods for Generating Hash Codes

In most applications, the keys that we will want to store in a table will consist of strings of letters or digits rather than a single character (e.g., a social security number, a person's name, or a part ID). We need a way to map each string to a particular table index. Again, we have a situation in which the number of possible key values is much larger than the table size. For example, if a string can store up to 10 letters or digits, the number of possible strings is 36^{10} (approximately 3.7×10^{15}), assuming the English alphabet with 26 letters.

Generating good hash codes for arbitrary strings or arbitrary objects is somewhat of an experimental process. Simple algorithms tend to generate a lot of collisions. For example, simply summing the `int` values for all characters in a string would generate the same hash code for words that contained the same letters but in different orders, such as "sign" and "sing", which would have the same hash code using this algorithm `('s' + 'i' + 'n' + 'g')`. The algorithm used by the Java API accounts for the position of the characters in the string as well as the character values.

The `String.hashCode()` method returns the integer calculated by the formula:

$$s_0 \times 31^{(n-1)} + s_1 \times 31^{(n-2)} + \ldots + s_{n-1}$$

where s_i is the ith character of the string and `n` is the length of the string. For example, the string `"Cat"` would have a hash code of `'C' × 31² + 'a' × 31 + 't'`. This is the number 67,510. (The number 31 is a prime number that generates relatively few collisions.)

As previously discussed, the integer value returned by method `String.hashCode` can't be unique because there are too many possible strings. However, the probability of two strings having the same hash code value is relatively small because the `String.hashCode` method distributes the hash code values fairly evenly throughout the range of **`int`** values.

Because the hash codes are distributed evenly throughout the range of **`int`** values, method `String.hashCode` will appear to produce a random value, as will the expressions `s.hashCode() % table.length`, which selects the initial value of `index` for `String s`. If the object is not already present in the table, the probability that this expression does not yield an empty slot in the table is proportional to how full the table is.

One additional criterion for a good hash function, besides a random distribution for its values, is that it be relatively simple and efficient to compute. It doesn't make much sense to use a hash function whose computation is an O(n) process to avoid doing an O(n) search.

Open Addressing

Next, we consider two ways to organize hash tables: open addressing and chaining. In open addressing, each hash table element (`type Object`) references a single key–value pair. We can use the following simple approach (called *linear probing)* to access an item in a hash table. If the index calculated for an item's key is occupied by an item with that key, we have found the item. If that element contains an item with a different key, we increment the index by 1. We keep incrementing the index (modulo the table length) until either we find

the key we are seeking or we reach a `null` entry. A `null` entry indicates that the key is not in the table.

Algorithm for Searching for an Item in a Hash Table

1. Compute the index by taking the item's `hashCode() % table.length`.
2. `if table [index]` is `null`
3. The item is not in the table.
4. `else if table [index]` is equal to the item
5. The item is in the table.

   `else`
6. Continue to search the table by incrementing the index until either the item is found or a `null` entry is found.

Step 1 ensures that the `index` is within the table range (0 through `table.length - 1`). If the condition in Step 2 is `true`, the table index does not reference an object, so the item is not in the table. The condition in Step 4 is `true` if the item being sought is at position index, in which case the item is located. Steps 1 through 5 can be done in O(1) expected time.

Step 6 is necessary for two reasons. The values returned by method `hashCode` are not unique, so the item being sought can have the same hash code as another one in the table. Also, the remainder calculated in Step 1 can yield the same index for different hash code values. Both of these cases are examples of collisions.

Table Wraparound and Search Termination

Note that as you increment the table index, your table should wrap around (as in a circular array) so that the element with subscript 0 "follows" the element with subscript `table.length - 1`. This enables you to use the entire table, not just the part with subscripts larger than the hash code value, but it leads to the potential for an infinite loop in Step 6 of the algorithm. If the table is full and the objects examined so far do not match the one you are seeking, how do you know when to stop? One approach would be to stop when the index value for the next probe is the same as the hash code value for the object. This means that you have come full circle to the starting value for the index. A second approach would be to ensure that the table is never full by increasing its size after an insertion if its occupancy rate exceeds a specified threshold. This is the approach that we take in our implementation.

EXAMPLE 7.8 We illustrate insertion of five names in a table of size 5 and in a table of size 11. Table 7.3 shows the names, the corresponding hash code, the hash code modulo 5 (in column 3), and the hash code modulo 11 (in column 4). We picked prime numbers (5 and 11) because empirical tests have shown that hash tables with a size that is a prime number often give better results.

For a table of size 5 (an occupancy rate of 100 percent), `"Tom"`, `"Dick"`, and `"Sam"` have hash indexes of 4, and `"Harry"` and `"Pete"` have hash indexes of 3; for a table length of 11 (an occupancy rate of 45 percent), `"Dick"` and `"Sam"` have hash indexes of 5, but the others have hash indexes that are unique. We see how the insertion process works next.

For a table of size 5, if `"Tom"` and `"Dick"` are the first two entries, `"Tom"` would be stored at the element with index 4, the last element in the table. Consequently, when `"Dick"` is inserted, because element 4 is already occupied, the hash index is incremented to 0 (the table wraps around to the beginning), where `"Dick"` is stored.

TABLE 7.3
Names and `hashCode` Values for Table Sizes 5 and 11

Name	hashCode()	hashCode()%5	hashCode()%11
"Tom"	84274	4	3
"Dick"	2129869	4	5
"Harry"	69496448	3	10
"Sam"	82879	4	5
"Pete"	2484038	3	7

[0]	"Dick"
[1]	**null**
[2]	**null**
[3]	**null**
[4]	"Tom"

"Harry" is stored in position 3 (the hash index), and "Sam" is stored in position 1 because its hash index is 4 but the elements at 4 and 0 are already filled.

[0]	"Dick"
[1]	"Sam"
[2]	**null**
[3]	"Harry"
[4]	"Tom"

Finally, "Pete" is stored in position 2 because its hash index is 3 but the elements at positions 3, 4, 0, 1 are filled.

[0]	"Dick"
[1]	"Sam"
[2]	"Pete"
[3]	"Harry"
[4]	"Tom"

For the table of size 11, the entries would be stored as shown in the following table, assuming that they were inserted in the order "Tom", "Dick", "Harry", "Sam", and finally "Pete". Insertions go more smoothly for the table of size 11. The first collision occurs when "Sam" is stored, so "Sam" is stored at position 6 instead of position 5.

[0]	**null**
[1]	**null**
[2]	**null**
[3]	"Tom"
[4]	**null**
[5]	"Dick"

[6]	"Sam"
[7]	"Pete"
[8]	null
[9]	null
[10]	"Harry"

For the table of size 5, retrieval of "Tom" can be done in one step. Retrieval of all of the others would require a linear search because of collisions that occurred when they were inserted. For the table of size 11, retrieval of all but "Sam" can be done in one step, and retrieval of "Sam" requires only two steps. This example illustrates that the best way to reduce the probability of a collision is to increase the table size.

Traversing a Hash Table

One thing that you cannot do is traverse a hash table in a meaningful way. If you visit the hash table elements in sequence and display the objects stored, you would display the strings "Dick", "Sam", "Pete", "Harry", and "Tom" for the table of length 5 and the strings "Tom", "Dick", "Sam", "Pete", and "Harry" for a table of length 11. In either case, the list of names is in arbitrary order.

Deleting an Item Using Open Addressing

When an item is deleted, we cannot just set its table entry to **null**. If we do, then when we search for an item that may have collided with the deleted item, we may incorrectly conclude that the item is not in the table. (Because the item that collided was inserted after the deleted item, we will have stopped our search prematurely.) By storing a dummy value when an item is deleted, we force the search algorithm to keep looking until either the desired item is found or a **null** value, representing a free cell, is located.

Although the use of a dummy value solves the problem, keep in mind that it can lead to search inefficiency, particularly when there are many deletions. Removing items from the table does not reduce the search time because the dummy value is still in the table and is part of a search chain. In fact, you cannot even replace a deleted value with a new item because you still need to go to the end of the search chain to ensure that the new item is not already present in the table. So deleted items waste storage space and reduce search efficiency. In the worst case, if the table is almost full and then most of the items are deleted, you will have $O(n)$ performance when searching for the few items remaining in the table.

Reducing Collisions by Expanding the Table Size

Even with a good hashing function, it is still possible to have collisions. The first step in reducing these collisions is to use a prime number for the size of the table.

In addition, the probability of a collision is proportional to how full the table is. Therefore, when the hash table becomes sufficiently full, a larger table should be allocated and the entries reinserted.

We previously saw examples of expanding the size of an array. Generally, what we did was to allocate a new array with twice the capacity of the original, copy the values in the original array to the new array, and then reference the new array instead of the original. This approach will not work with hash tables. If you use it, some search chains will be broken

because the new table does not wrap around in the same way as the original table. The last element in the original table will be in the middle of the new table, and it does not wrap around to the first element of the new table. Therefore, you expand a hash table (called *rehashing*) using the following algorithm.

Algorithm for rehashing

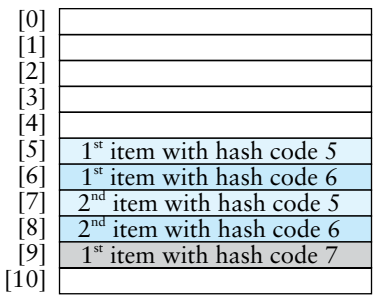
1. Allocate a new hash table with twice the capacity of the original.
2. Reinsert each old table entry that has not been deleted into the new hash table.
3. Reference the new table instead of the original.

Step 2 reinserts each item from the old table into the new table instead of copying it over to the same location. We illustrate this in the hash table implementation. Note that deleted items are not reinserted into the new table, thereby saving space and reducing the length of some search chains.

Reducing Collisions Using Quadratic Probing

The problem with linear probing is that it tends to form clusters of keys in the table, causing longer search chains. For example, if the table already has keys with hash codes of 5 and 6, a new item that collides with either of these keys will be placed at index 7. An item that collides with any of these three items will be placed at index 8, and so on. Figure 7.5 shows a hash table of size 11 after inserting elements with hash codes in the sequence 5, 6, 5, 6, 7. Each new collision expands the cluster by one element, thereby increasing the length of the search chain for each element in that cluster. For example, if another element is inserted with any hash code in the range 5 through 9, it will be placed at position 10, and the search chain for items with hash codes of 5 and 6 would include the elements at indexes 7, 8, 9, and 10.

FIGURE 7.5
Clustering with Linear Probing



One approach to reduce the effect of clustering is to use *quadratic probing* instead of linear probing. In quadratic probing, the increments form a quadratic series ($1 + 2^2 + 3^2 + \cdots$). Therefore, the next value of `index` is calculated using the steps:

```
probeNum++;
index = (startIndex + probeNum * probeNum) % table.length
```

where `startIndex` is the index calculated using method `hashCode` and `probeNum` starts at 0. Ignoring wraparound, if an item has a hash code of 5, successive values of `index` will be 6 (5 +1), 9 (5 + 4), 14 (5 + 9), . . ., instead of 6, 7, 8, ... Similarly, if the hash code is 6, successive values of `index` will be 7, 10, 15, and so on. Unlike linear probing, these two search chains have only one table element in common (at index 6).

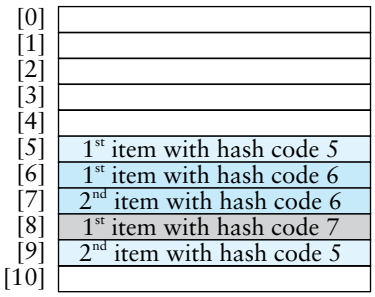


FIGURE 7.6
Insertion with Quadratic Probing

Figure 7.6 illustrates the hash table after elements with hash codes in the same sequence as in the preceding table (5, 6, 5, 6, 7) have been inserted with quadratic probing. Although the cluster of elements looks similar, their search chains do not overlap as much as before. Now the search chain for an item with a hash code of 5 consists of the elements at 5, 6, and 9, and the search chain for an item with a hash code of 6 consists of the elements at positions 6 and 7.

Problems with Quadratic Probing

One disadvantage of quadratic probing is that the next index calculation is a bit time-consuming as it involves a multiplication, an addition, and a modulo division. A more efficient way to calculate the next index follows:

```
k += 2;
index = (index + k) % table.length;
```

which replaces the multiplication with an addition. If the initial value of `k` is –1, successive values of `k` will be 1, 3, 5, 7, ... If the hash code is 5, successive values of `index` will be 5, 6 (5 + 1), 9 (5 + 1 + 3), 14 (5 + 1 + 3 + 5),. . . . The proof of the equality of these two approaches to calculating `index` is based on the following mathematical series:

$$n^2 = 1 + 3 + 5 + \cdots + 2n - 1$$

A more serious problem with quadratic probing is that not all table elements are examined when looking for an insertion index, so it is possible that an item can't be inserted even when the table is not full. It is also possible that your program can get stuck in an infinite loop while searching for an empty slot. It can be proved that if the table size is a prime number and the table is never more than half full, this can't happen. However, requiring that the table be half empty at all times wastes quite a bit of memory. For these reasons, we will use linear probing in our implementation.

Chaining

An alternative to open addressing is a technique called *chaining*, in which each table element references a linked list that contains all the items that hash to the same table index. This linked list is often called a *bucket*, and this approach is sometimes called *bucket hashing*. Figure 7.7 shows the result of chaining for our earlier example with a table of size 5. Each new element with a particular hash index can be placed at the beginning or the end of the associated linked list. The algorithm for accessing such a table is the same as for open addressing, except for the step for resolving collisions. Instead of incrementing the table index to access the next item with a particular hash code value, you traverse the linked list referenced by the table element with index `hashCode() % table.length`.

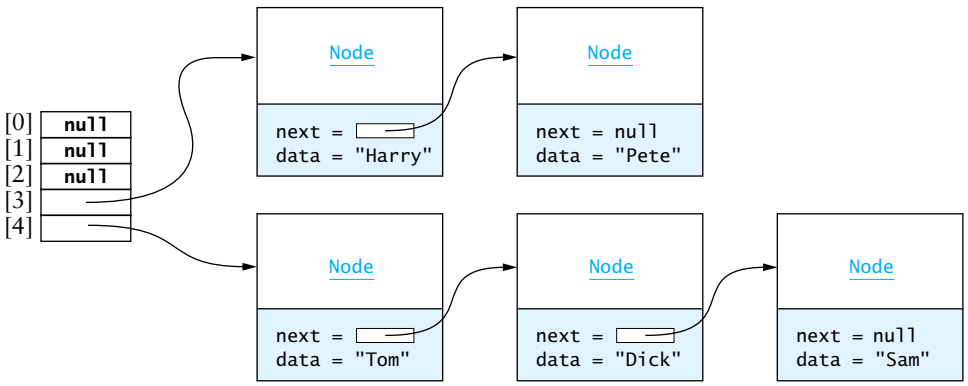


FIGURE 7.7
Example of Chaining

One advantage of chaining is that only items that have the same value for `hashCode() % table.length` will be examined when looking for an object. In open addressing, search chains can overlap, so a search chain may include items in the table that have different starting index values.

A second advantage is that you can store more elements in the table than the number of table slots (indexes), which is not the case for open addressing. If each table index already references a linked list, additional items can be inserted in an existing list without increasing the table size (number of indexes).

Once you have determined that an item is not present, you can insert it either at the beginning or at the end of the list. To delete an item, simply remove it from the list. In contrast to open addressing, removing an item actually deletes it, so it will not be part of future search chains.

Performance of Hash Tables

The *load factor* for a hash table is the number of filled cells divided by table size. The load factor has the greatest effect on hash table performance. The lower the load factor, the better the performance because there is less chance of a collision when a table is sparsely populated. If there are no collisions, the performance for search and retrieval is O(1), regardless of the table size.

Performance of Open Addressing versus Chaining

Donald E. Knuth (*Searching and Sorting*, vol. 3 of *The Art of Computer Programming*, Addison-Wesley, 1973) derived the following formula for the expected number of comparisons, c, required for finding an item that is in a hash table using open addressing with linear probing and a load factor L:

$$c = \frac{1}{2}\left(1 + \frac{1}{1 - L}\right)$$

Table 7.4 (second column) shows the value of c for different values of load factor (L). It shows that if L is 0.5 (half full), the expected number of comparisons required is 1.5. If L increases to 0.75, the expected number of comparisons is 2.5, which is still very respectable. If L increases to 0.9 (90 percent full), the expected number of comparisons is 5.5. This is **`true`** regardless of the size of the table.

TABLE 7.4
Number of Probes for Different Values of Load Factor (*L*)

L	Number of Probes with Linear Probing	Number of Probes with Chaining
0.0	1.00	1.00
0.25	1.17	1.13
0.5	1.50	1.25
0.75	2.50	1.38
0.85	3.83	1.43
0.9	5.50	1.45
0.95	10.50	1.48

Using chaining, if an item is in the table, on average we have to examine the table element corresponding to the item's hash code and then half of the items in each list. The average number of items in a list is L, the number of items divided by the table size. Therefore, we get the formula

$$c = 1 + \frac{L}{2}$$

for a successful search. Table 7.4 (third column) shows the results for chaining. For values of L between 0.0 and 0.75, the results are similar to those of linear probing, but chaining gives better performance than linear probing for higher load factors. Quadratic probing (not shown) gives performance that is between those of linear probing and chaining.

Performance of Hash Tables versus Sorted Arrays and Binary Search Trees

If we compare hash table performance with binary search of a sorted array, the number of comparisons required by binary search is O(log n), so the number of comparisons increases with table size. A sorted array of size 128 would require up to 7 probes (2^7 is 128), which is more than for a hash table of any size that is 90 percent full. A sorted array of size 1024 would require up to 10 probes (2^{10} is 1024). A binary search tree would yield the same results.

You can insert into or remove elements from a hash table in O(1) expected time. Insertion or removal from a binary search tree is O(log n), but insertion or removal from a sorted array is O(n) (you need to shift the larger elements over). (Worst-case performance for a hash table or a binary search tree is O(n).)

Storage Requirements for Hash Tables, Sorted Arrays, and Trees

The performance of hashing is certainly preferable to that of binary search of an array (or a binary search tree), particularly if L is less than 0.75. However, the tradeoff is that the lower the load factor, the more unfilled storage cells there are in a hash table, whereas there are no empty cells in a sorted array. Because a binary search tree requires three references per

node (the item, the left subtree, and the right subtrees), more storage would be required for a binary search tree than for a hash table with a load factor of 0.75.

EXAMPLE 7.9 A hash table of size 100 with open addressing could store 75 items with a load factor of 0.75. This would require storage for 100 references. This would require storage for 100 references (25 references would be `null`).

Storage requirements for Open Addressing and Chaining

Next, we consider the effect of chaining on storage requirements. For a table with a load factor of L, the number of table elements required is n (the size of the table). For open addressing, the number of references to an item (a key–value pair) is n. For chaining, the average number of nodes in a list is L. If we use the Java API `LinkedList`, there will be three references in each node (the item, the next list element, and the previous element). However, we could use our own single-linked list and eliminate the previous-element reference (at some time cost for deletions). Therefore, we will require storage for $n + 2L$ references.

EXAMPLE 7.10 If we have 60,000 items in our hash table and use open addressing, we would need a table size of 80,000 to have a load factor of 0.75 and an expected number of comparisons of 2.5. Next, we calculate the table size, n, needed to get similar performance using chaining.

$$
\begin{aligned}
2.5 &= 1 + \frac{L}{2} \\
5.0 &= 2 + L \\
3.0 &= \frac{60{,}000}{n} \\
n &= 20{,}000
\end{aligned}
$$

A hash table of size 20,000 requires storage space for 20,000 references to lists. There will be 60,000 nodes in the table (one for each item). If we use linked lists of nodes, we will need storage for 140,000 references (2 references per node plus the 20,000 table references). This is almost twice the storage needed for open addressing.

EXERCISES FOR SECTION 7.3

SELF-CHECK

1. For the hash table search algorithm shown in this section, why was it unnecessary to test whether all table entries had been examined as part of Step 5?
2. For the items in the five-element table of Table 7.3, compute `hashCode() % table.length` for lengths of 7 and 13. What would be the position of each word in tables of these sizes using open addressing and linear probing? Answer the same question for chaining.

3. The following table stores `Integer` keys with the **`int`** values shown. Show one sequence of insertions that would store the keys as shown. Which elements were placed in their current position because of collisions? Show the table that would be formed by chaining.

Index	Key
[0]	24
[1]	6
[2]	20
[3]	
[4]	14

4. For Table 7.3 and the table size of 5 shown in Example 7.8, discuss the effect of deleting the entry for Dick and replacing it with a **`null`** value. How would this affect the search for Sam, Pete, and Harry? Answer both questions if you replace the entry for Dick with the string `"deleted"` instead of **`null`**.

5. Explain what is wrong with the following strategy to reclaim space that is filled with deleted items in a hash table: when attempting to insert a new item in the table, if you encounter an item that has been deleted, replace the deleted item with the new item.

6. Compare the storage requirement for a hash table with open addressing, a table size of 500, and a load factor of 0.5 with a hash table that uses chaining and gives the same performance.

7. One simple hash code is to use the sum of the ASCII codes for the letters in a word. Explain why this is not a good hash code.

8. If p_i is the position of a character in a string and c_i is the code for that character, would $c_1p_1 + c_2p_2 + c_3p_3 + \ldots$ be a better hash code? Explain why or why not.

9. Use the hash code in Self-Check Exercise 7 to store the words `"cat"`, `"hat"`, `"tac"`, and `"act"` in a hash table of size 10. Show this table using open hashing and chaining.

PROGRAMMING

1. Code the following algorithm for finding the location of an object as a static method. Assume a hash table array and an object to be located in the table are passed as arguments. Return the object's position if it is found; return –1 if the object is not found.

    1. Compute the index by taking the `hashCode() % table.length`.
    2. **`if`** `table[index]` is **`null`**
    3. The object is not in the table. Return -1.

       `else if` `table[index]` is equal to the object
    4. The object is in the table. Return `index`.

       `else`
    5. Continue to search the table (by incrementing `index`) until either the object is found or a **`null`** entry is found.

7.4 Implementing the Hash Table

In this section, we discuss how to implement a hash table. We will show implementations for hash tables using open addressing and chaining.

Interface KWHashMap

Because we want to show more than one way to implement a hash table, we introduce an interface `KWHashMap<K, V>` in Table 7.5. The methods for interface `KWHashMap<K, V>` (`get`, `put`, `isEmpty`, `remove`, and `size`) are similar to the ones shown earlier for the `Map` interface (see Table 7.2). There is a class `Hashtable` in the Java API `java.util`; however, it has been superseded by the class `HashMap`. Our interface `KWHashMap` doesn't include all the methods of interface `Map`.

Class Entry

A hash table stores key–value pairs, so we will use an inner class `Entry` in each hash table implementation with data fields `key` and `value` (see Table 7.6). The implementation of inner class `Entry` is straightforward, and we show it in Listing 7.3.

TABLE 7.5
Interface `KWHashMap<K, V>`

Method	behavior
`V get(Object key)`	Returns the value associated with the specified `key`. Returns **`null`** if the `key` is not present
`boolean isEmpty()`	Returns **`true`** if this table contains no `key-value` mappings
`V put(K key, V value)`	Associates the specified `value` with the specified `key`. Returns the previous `value` associated with the specified `key`, or **`null`** if there was no mapping for the `key`
`V remove(Object key)`	Removes the mapping for this `key` from this table if it is present (optional operation). Returns the previous `value` associated with the specified `key`, or **`null`** if there was no mapping
`int size()`	Returns the size of the table

TABLE 7.6
Inner Class `Entry<K, V>`

Data Field	Attribute
`private finalK key`	The `key`
`private V value`	The `value`
Constructor	**behavior**
`public Entry(K key, V value)`	Constructs an `Entry` with the given `value`
Method	**behavior**
`public K getKey()`	Retrieves the `key`
`public V getValue()`	Retrieves the `value`
`public V setValue(V val)`	Sets the `value`

LISTING 7.3
Inner Class `Entry`

```
/** Contains key-value pairs for a hash table. */
private static class Entry<K, V> {

    /** The key */
    private final K key;
    /** The value */
    private V value;

    /** Creates a new key-value pair.
        @param key The key
        @param value The value
     */
    public Entry(K key, V value) {
        this.key = key;
        this.value = value;
    }

    /** Retrieves the key.
        @return The key
     */
    public K getKey() {
        return key;
    }

    /** Retrieves the value.
        @return The value
     */
    public V getValue() {
        return value;
    }

    /** Sets the value.
        @param val The new value
        @return The old value
     */
    public V setValue(V val) {
        V oldVal = value;
        value = val;
        return oldVal;
    }

    @Override
    /** Return a string representation of this entry
        @return "(key, value)"
     */
    public String toString() {
        return "(" + key + ", " + value + ")";
    }
}
```

Class HashtableOpen

In a hash table that uses open addressing, we represent the hash table as an array of `Entry` objects (initial size is `START_CAPACITY`). We describe the data fields in Table 7.7. The `Entry` object `DELETED` is used to indicate that the `Entry` at a particular table element has been deleted; a **`null`** reference indicates that a table element was never occupied.

The data field declarations and constructor for `HashtableOpen` follow. Because generic arrays are not permitted, the constructor creates an `Entry[]` array, which is referenced by `table` (type `Entry<K, V>[]`).

```
/** Hash table implementation using open addressing. */
public class HashtableOpen<K, V> implements KWHashMap<K, V> {

    // Insert inner class Entry<K, V> here.
    // Data Fields
    private Entry<K, V>[] table;
    private static final int START_CAPACITY = 101;
    private double LOAD_THRESHOLD = 0.75;
    private int numKeys;
    private int numDeletes;
    private final Entry<K, V> DELETED =
            new Entry<>(null, null);

    // Constructor
    public HashTableOpen() {
        table = new Entry[START_CAPACITY];
    }
    . . .
```

TABLE 7.7
Data Fields for Class `HashtableOpen<K, V>`

Data Field	Attribute
`private Entry<K, V>[] table`	The hash table array
`private static final int START_CAPACITY`	The initial capacity
`private double LOAD_THRESHOLD`	The maximum load factor
`private int numKeys`	The number of keys in the table excluding keys that were deleted
`private int numDeletes`	The number of deleted keys
`private final Entry<K, V> DELETED`	A special object to indicate that an entry has been deleted

Several methods for class `HashtableOpen` use a private method `find` that searches the table (using linear probing) until it finds either the target key or an empty slot. By expanding the table when its load factor exceeds the `LOAD_THRESHOLD`, we ensure that there will always be an empty slot in the table. Table 7.8 summarizes these private methods.

TABLE 7.8
Private Methods for Class `HashtableOpen`

Method	behavior
`private int find(Object key)`	Returns the index of the specified `key` if present in the table; otherwise, returns the index of the first available slot
`private void rehash()`	Doubles the capacity of the table and permanently removes deleted items

The algorithm for method `find` follows. Listing 7.4 shows the method.

Algorithm for HashtableOpen.find(Object key)

1. Set `index` to `key.hashCode() % table.length`.
2. **`if`** `index` is negative, add `table.length`.
3. **`while`** `table[index]` is not empty and the key is not at `table[index]`
4. Increment `index`.

5. **if** index is greater than or equal to table.length
6. Set index to 0.
7. Return the index.

LISTING 7.4
Method HashtableOpen.find

```
/** Finds either the target key or the first empty slot in the
    search chain using linear probing.
    pre: The table is not full.
    @param key The key of the target object
    @return The position of the target or the first empty slot if
            the target is not in the table.
 */
private int find(Object key) {
    // Calculate the starting index.
    int index = key.hashCode() % table.length;
    if (index < 0)
        index += table.length; // Make it positive.

    // Increment index until an empty slot is reached
    // or the key is found.
    while ((table[index] != null)
            && (!key.equals(table[index].getKey()))) {
        index++;
        // Check for wraparound.
        if (index >= table.length)
            index = 0; // Wrap around.
    }
    return index;
}
```

Note that the method call key.hashCode() calls key's hashCode. The condition (!key.equals (table[index].getKey())) compares the key at table[index] with the key being sought (the method parameter).

Next, we discuss the public methods: get and put. Listing 7.5 shows the code. The get algorithm follows.

Algorithm for get(Object key)

1. Find the first table element that is empty or the table element that contains the key.
2. **if** the table element found contains the key
   Return the value at this table element.
3. **else**
4. Return **null**.

LISTING 7.5
Method HashtableOpen.get

```
/** Method get for class HashtableOpen.
    @param key The key being sought
    @return the value associated with this key if found; otherwise, null
 */
@Override
public V get(Object key) {
    // Find the first table element that is empty
    // or the table element that contains the key.
```

```
    int index = find(key);
    // If the search is successful, return the value.
    if (table[index] != null)
        return table[index].getValue();
    else
        return null; // key not found.
}
```

Next, we write the algorithm for method `put`. After inserting a new entry, the method checks to see whether the load factor exceeds the `LOAD_THRESHOLD`. If so, it calls method `rehash` to expand the table and reinsert the entries. Listing 7.6 shows the code for method `put`.

Algorithm for HashtableOpen.put(K key,V value)

1. Find the first table element that is empty or the table element that contains the `key`.
2. **`if`** an empty element was found
3. Insert the new item and increment `numKeys`.
4. Check for need to rehash.
5. Return **`null`**.
6. The `key` was found. Replace the `value` associated with this table element and return the old value.

LISTING 7.6

Method `HashtableOpen.put`

```
/** Method put for class HashtableOpen.
    post: This key-value pair is inserted in the
          table and numKeys is incremented. If the key is already
          in the table, its value is changed to the argument
          value and numKeys is not changed. If the LOAD_THRESHOLD
          is exceeded, the table is expanded.
    @param key The key of item being inserted
    @param value The value for this key
    @return Old value associated with this key if found;
            otherwise, null
 */
@Override
public V put(K key, V value) {
    // Find the first table element that is empty
    // or the table element that contains the key.
    int index = find(key);

    // If an empty element was found, insert new entry.
    if (table[index] == null) {
        table[index] = new Entry<>(key, value);
        numKeys++;
        // Check whether rehash is needed.
        double loadFactor =
                (double) (numKeys + numDeletes) / table.length;
        if (loadFactor > LOAD_THRESHOLD)
            rehash();
        return null;
    }

    // assert: table element that contains the key was found.
    // Replace value for this key.
    V oldVal = table[index].getValue();
    table[index].setValue(value);
    return oldVal;
}
```

PITFALL

Integer Division for Calculating Load Factor

Before calling method `rehash`, method `put` calculates the load factor by dividing the number of filled slots by the table size. This is a simple computation, but if you forget to cast the numerator or denominator to **`double`**, the load factor will be zero (because of integer division), and the table will not be expanded. This will slow down the performance of the table when it becomes nearly full, and it will cause an infinite loop (in method `find`) when the table is completely filled.

Next, we write the algorithm for method `remove`. Note that we "remove" a table element by setting it to reference object `DELETED`. We leave the implementation as an exercise.

Algorithm for remove (Object key)

1. Find the first table element that is empty or the table element that contains the `key`.
2. **`if`** an empty element was found
3. Return **`null`**.
4. `key` was found. Remove this table element by setting it to reference `DELETED`, increment `numDeletes`, and decrement `numKeys`.
5. Return the `value` associated with this `key`.

Finally, we write the algorithm for private method `rehash`. Listing 7.7 shows the method. Although we do not take the effort to make the table size a prime number, we do make it an odd number.

Algorithm for HashtableOpen.rehash

1. Allocate a new hash table that is double the size and has an odd length.
2. Reset the number of keys and number of deletions to 0.
3. Reinsert each table entry that has not been deleted in the new hash table.

LISTING 7.7

Method `HashtableOpen.rehash`

```
/** Expands table size when loadFactor exceeds LOAD_THRESHOLD
    post: The size of the table is doubled and is an odd integer.
          Each nondeleted entry from the original table is
          reinserted into the expanded table.
          The value of numKeys is reset to the number of items
          actually inserted; numDeletes is reset to 0.
 */
private void rehash() {
    // Save a reference to oldTable.
    Entry<K, V>[] oldTable = table;
    // Double capacity of this table.
    table = new Entry[2 * oldTable.length + 1];

    // Reinsert all items in oldTable into expanded table.
    numKeys = 0;
    numDeletes = 0;
    for (int i = 0; i < oldTable.length; i++) {
        if ((oldTable[i] != null) && (oldTable[i] != DELETED)) {
            // Insert entry in expanded table
            put(oldTable[i].getKey(), oldTable[i].getValue());
        }
    }
}
```

TABLE 7.9
Data Fields for Class `HashtableChain<K, V>`

Data Field	Attribute
`private LinkedList<Entry<K, V>>[] table`	A table of references to linked lists of `Entry<K, V>` objects
`private int numKeys`	The number of keys (entries) in the table
`private static final int CAPACITY`	The size of the table
`private static final int LOAD_THRESHOLD`	The maximum load factor

Class HashtableChain

Next, we turn our attention to class `HashtableChain`, which implements `KWHashMap` using chaining. We will represent the hash table as an array of linked lists as shown in Table 7.9. Even though a hash table that uses chaining can store any number of elements in the same slot, we will expand the table if the number of entries becomes three times the number of slots (`LOAD_THRESHOLD` is 3.0) to keep the performance at a reasonable level.

PROGRAM STYLE

It is generally preferred that data fields be defined as interfaces and that implementing classes are assigned by the constructor. However, in this case, we define the data field `table` to be a `LinkedList`. This is because we want a linked list and use a method (`addFirst`) that is defined in the `LinkedList` class, but not in the `List` interface.

Listing 7.8 shows the data fields and the constructor for class `HashtableChain`.

LISTING 7.8
Data Fields and Constructor for `HashtableChain.java`

```
import java.util.*;

/** Hash table implementation using chaining. */
public class HashtableChain<K, V> implements KWHashMap<K, V> {

    // Insert inner class Entry<K, V> here.
    /** The table */
    private LinkedList<Entry<K, V>>[] table;
    /** The number of keys */
    private int numKeys;
    /** The capacity */
    private static final int CAPACITY = 101;
    /** The maximum load factor */
    private static final double LOAD_THRESHOLD = 3.0;

    // Constructor
    public HashtableChain() {
        table = new LinkedList[CAPACITY];
    }
    . . .
}
```

Next, we discuss the three methods `get`, `put`, and `remove`. Instead of introducing a `find` method to search a list for the key, we will include a search loop in each method. We will create a `ListIterator` object and use that object to access each list element.

We begin with the algorithm for get. Listing 7.9 shows its code. We didn't use methods getKey and getValue to access an item's key and value because those private data fields of class Entry are visible in the class that contains it.

Algorithm for HashtableChain.get (Object key)

1. Set index to key.hashCode() % table.length.
2. if index is negative
3. Add table.length.
4. if table[index] is null
5. key is not in the table; return null.
6. For each element in the list at table[index]
7. if that element's key matches the search key
8. Return that element's value.
9. key is not in the table; return null.

LISTING 7.9

Method HashtableChain.get

```
/** Method get for class HashtableChain.
    @param key The key being sought
    @return The value associated with this key if found;
            otherwise, null
 */
@Override
public V get(Object key) {
    int index = key.hashCode() % table.length;
    if (index < 0)
        index += table.length;
    if (table[index] == null)
        return null; // key is not in the table.

    // Search the list at table[index] to find the key.
    for (Entry<K, V> nextItem : table[index]) {
        if (nextItem.getKey().equals(key))
            return nextItem.getValue();
    }

    // assert: key is not in the table.
    return null;
}
```

Next, we write the algorithm for method put. Listing 7.10 shows its code.

Algorithm for HashtableChain.put (K key,V value)

1. Set index to key.hashCode() % table.length.
2. if index is negative, add table.length.
3. if table[index] is null
4. Create a new linked list at table[index].
5. Search the list at table[index] to find the key.
6. if the search is successful
7. Replace the value associated with this key.
8. Return the old value.
9. Insert the new key–value pair in the linked list at table[index]
10. Increment numKeys.

11. if the load factor exceeds the LOAD_THRESHOLD
12. Rehash.
13. Return **null**.

LISTING 7.10
Method HashtableChain.put

```
/** Method put for class HashtableChain.
    post: This key-value pair is inserted in the
          table and numKeys is incremented. If the key is already
          in the table, its value is changed to the argument
          value and numKeys is not changed.
    @param key The key of item being inserted
    @param value The value for this key
    @return The old value associated with this key if
            found; otherwise, null
*/
@Override
public V put(K key, V value) {
    int index = key.hashCode() % table.length;
    if (index < 0)
        index += table.length;
    if (table[index] == null) {
        // Create a new linked list at table[index].
        table[index] = new LinkedList<>();
    }

    // Search the list at table[index] to find the key.
    for (Entry<K, V> nextItem : table[index]) {
        // If the search is successful, replace the old value.
        if (nextItem.getKey().equals(key)) {
            // Replace value for this key.
            V oldVal = nextItem.getValue();
            nextItem.setValue(value);
            return oldVal;
        }
    }

    // assert: key is not in the table, add new item.
    table[index].addFirst(new Entry<>(key, value));
    numKeys++;
    if (numKeys > (LOAD_THRESHOLD * table.length))
        rehash();
    return null;
}
```

Last, we write the algorithm for method remove. We leave the implementation of rehash and remove as an exercise.

Algorithm for HashtableChain.remove(Object key)

1. Set index to key.hashCode() % table.length.
2. **if** index is negative, add table.length.
3. **if** table[index] is **null**
4. key is not in the table; return **null**.
5. Search the list at table[index] to find the key.

6. if the search is successful
7. Remove the entry with this key and decrement numKeys.
8. if the list at table[index] is empty
9. Set table[index] to null.
10. Return the value associated with this key.
11. The key is not in the table; return null.

Testing the Hash Table Implementations

We discuss two approaches to testing the hash table implementations. One way is to create a file of key–value pairs and then read each key–value pair and insert it in the hash table, observing how the table is filled. To do this, you need to write a toString method for the table that captures the index of each table element that is not null and then the contents of that table element. For open addressing, the contents would be the string representation of the key–value pair. For chaining, you could use a list iterator to traverse the linked list at that table element and append each key–value pair to the result string (see the Programming exercises for this section).

If you use a data file, you can carefully test different situations. The following are some of the cases you should examine:

- Does the array index wrap around as it should?
- Are collisions resolved correctly?
- Are duplicate keys handled appropriately? Is the new value retrieved instead of the original value?
- Are deleted keys retained in the table but no longer accessible via a get?
- Does rehashing occur when the load factor reaches 0.75 (3.0 for chaining)?

By stepping through the get and put methods, you can observe how the table is probed and examine the search chain that is followed to access or retrieve a key.

An alternative to creating a data file is to insert randomly generated integers in the hash table. This will allow you to create a very large table with little effort. The following loop generates SIZE key–value pairs. Each key is an integer between 0 and 32,000 and is autoboxed in an Integer object. For each table entry, the value is the same as the key. The Integer.hashCode method returns the int value of the object to which it is applied.

```
for (int i = 0; i < SIZE; i++) {
    Integer nextInt = (int) (32000 * Math.random());
    hashTable.put(nextInt, nextInt);
}
```

Because the keys are generated randomly, you can't investigate the effect of duplicate keys as you can with a data file. However, you can build arbitrarily large tables and observe how the elements are placed in the tables. After the table is complete, you can interactively enter items to retrieve, delete, and insert and verify that they are handled properly.

If you are using open addressing, you can add statements to count the number of items probed each time an insertion is made. You can accumulate these totals and display the average search chain length. If you are using chaining, you can also count the number of probes made and display the average. After all items have been inserted, you can calculate the average length of each linked list and compare that with the number predicted by the formula provided in the discussion of performance in Section 7.3.

EXERCISES FOR SECTION 7.4

SELF-CHECK

1. The following table stores `Integer` keys with the `int` values shown. Where would each key be placed in the new table resulting from rehashing the current table?

Index	Key
0	24
1	6
2	20
3	
4	14

PROGRAMMING

1. Write a `remove` method for class `HashtableOpen`.
2. Write `rehash` and `remove` methods for class `HashtableChain`.
3. Write a `toString` method for class `HashtableOpen`.
4. Write a `toString` method for class `HashtableChain`.
5. Write a method `size` for both hash table implementations.
6. Modify method `find` to count and display the number of probes made each time it is called. Accumulate these in a data field `numProbes` and count the number of times `find` is called in another data field. Provide a method that returns the average number of probes per call to `find`.

7.5 Implementation Considerations for Maps and Sets

Methods hashCode and equals

Class `Object` implements methods `hashCode` and `equals`, so every class can access these methods unless it overrides them. Method `Object.equals` compares two objects based on their addresses, not their contents. Similarly, method `Object.hashCode` calculates an object's hash code based on its address, not its contents. If you want to compare two objects for equality, you must implement an `equals` method for that class. In doing so, you should override the `equals` method for class `Object` by providing an `equals` method with the form

```
public boolean equals(Object obj) { . . . }
```

Most predefined classes (e.g., `String` and `Integer`) override method `equals` and method `hashCode`. If you override the `equals` method, Java recommends you also override the `hashCode` method. Otherwise, your class will violate the Java contract for `hashCode`, which states

> If `obj1.equals(obj2)` is **`true`**, then `obj1.hashCode() == obj2.hashCode()`.

Consequently, you should make sure that your `hashCode` method uses the same data field(s) as your `equals` method. We provide an example next.

EXAMPLE 7.11 Class `Person` has data field `IDNumber`, which is used to determine whether two `Person` objects are equal. The `equals` method returns **`true`** only if the objects' `IDNumber` fields have the same contents.

```
public boolean equals(Object obj) {
    if (obj instanceof Person)
        return IDNumber.equals(((Person) obj).IDNumber);
    else
        return false;
}
```

To satisfy its contract, method `Object.hashCode` must also be overridden as follows. Now two objects that are considered equal will also have the same hash code.

```
public int hashCode() {
    return IDNumber.hashCode();
}
```

Implementing HashSetOpen

Classes `HashtableOpen` and `HashtableChain` implement interface `KWHashMap` (our Map interface). Next we show how we might use these classes to implement the methods in the `Set` interface.

Table 7.10 compares corresponding `Map` and `Set` methods. The `Set` `contains` method performs a test for set membership instead of retrieving a value, so it is type **`boolean`**. Similarly, each of the other `Set` methods returns a **`boolean`** value that indicates whether the method was able to perform its task. The process of searching the hash table elements would be done the same way in each `Set` method as it is done in the corresponding `Map` method.

TABLE 7.10
Corresponding `Map` and `Set` Methods

Map Method	Set Method
`V get(Object key)`	`boolean contains(Object key)`
`V put(K key, V value)`	`boolean add(K key)`
`V remove(Object key)`	`boolean remove(Object key)`

For open addressing, method `put` uses the statement

```
table[index] = new Entry<>(key, value);
```

to store a reference to a new `Entry` object in the hash table. The corresponding statement in method `add` would be

```
table[index] = new Entry<>(key);
```

because the `key` is the only data that is stored.

Writing HashSetOpen as an Adapter Class

Instead of writing new `Set` methods from scratch, we can implement `HashSetOpen` as an adapter class with a `KWHashMap` data field implemented by a `HashtableOpen` object.

```
private final KWHashMap<K, K> setMap = new HashtableOpen<>();
```

We can write methods `contains`, `add`, and `remove` as follows. Because the map stores key–value pairs, we will have each set element reference an `Entry` object with the same key and value.

```
/** A hash table for storing set elements using open addressing. */
public class HashSetOpen<K> {
    private final KWHashMap<K, K> setMap = new HashtableOpen<>();
```

```
    /** Adapter method contains.
        @return true if the key is found in setMap
     */
    public boolean contains(Object key) {
        // HashtableOpen.get returns null if the key is not found.
        return (setMap.get(key) != null);
    }

    /** Adapter method add.
        post: Adds a new Entry object (key, key)
              if key is not a duplicate.
        @return true if the key is not a duplicate
     */
    public boolean add(K key) {
        /** HashtableOpen.put returns null if the
            key is not a duplicate. */
        return (setMap.put(key, key) == null);
    }

    /** Adapter method remove.
        post: Removes the key-value pair (key, key).
        @return true if the key is found and removed
     */
    public boolean remove(Object key) {
        /* HashtableOpen.remove returns null if the
           key is not removed. */
        return (setMap.remove(key) != null);
    }
}
```

Implementing the Java `Map` and `Set` Interfaces

Our goal in this chapter was to show you how to implement the operators in our hash table interface, not to implement the `Map` or `Set` interface fully. However, the Java API uses a hash table to implement both the `Map` and `Set` interfaces (class `HashMap` and class `HashSet`). You may be wondering what additional work would be required to implement the `Map` and `Set` interfaces using the classes we have developed so far.

The task of implementing these interfaces is simplified by the inclusion of abstract classes `AbstractMap` and `AbstractSet` in the Collections framework (see Figures 7.1 and 7.3). These classes provide implementations of several methods for the `Map` and `Set` interfaces. So, if class `HashtableOpen` extends class `AbstractMap`, we can reduce the amount of additional work we need to do. We should also replace `KWHashMap` with `Map`. Thus, the declaration for `HashtableOpen` would be class `HashtableOpen<K, V> extends AbstractMap<K, V> implements Map<K, V>`.

The `AbstractMap` provides relatively inefficient (O(n)) implementations of the `get` and `put` methods. Because we overrode these methods in both our implementations (`HashtableOpen` and `HashtableChain`), we will get O(1) expected performance. There are other, less critical methods that we don't need to provide because they are implemented in `AbstractMap` or its superclasses, such as `clear`, `isEmpty`, `putAll`, `equals`, `hashCode`, and `toString`.

Interface `Map.Entry` and Class `AbstractMap.SimpleEntry`

One requirement on the key–value pairs for a `Map` object is that they implement the interface `Map.Entry<K, V>`, which is an inner interface of interface `Map`. This may sound a bit confusing, but what it means is that an implementer of the `Map` interface must contain an inner class that provides code for the methods described in Table 7.11. The `AbstractMap` includes the inner class `SimpleEntry` that implements the `Map.Entry` interface. We can remove the inner class `Entry<K, V>` (Listing 7.3) and replace `new Entry` with `new SimpleEntry`.

TABLE 7.11
The `java.util.Map.Entry<K, V>` Interface

Method	Behavior
`K getKey()`	Returns the key corresponding to this entry
`V getValue()`	Returns the value corresponding to this entry
`V setValue(V val)`	Resets the value field for this entry to `val`. Returns its previous value field

Creating a Set View of a `Map`

Method `entrySet` creates a set view of the entries in a `Map`. This means that method `entrySet` returns an object that implements the `Set` interface—that is, a set. The members of the set returned are the key–value pairs defined for that `Map` object. For example, if a key is `"0123"` and the corresponding value is `"Jane Doe"`, the pair `("0123", "Jane Doe")` would be an element of the set view. This is called a view because it provides an alternative way to access the contents of the `Map`, but there is only a single copy of the underlying `Map` object.

We usually call method `entrySet` via a statement of the form:

```
Iterator<Map.Entry<K, V>> iter = myMap.entrySet().iterator();
```

The method call `myMap.entrySet()` creates a set view of `myMap`; next, we apply method `iterator` to that set, thereby returning an `Iterator` object for it. We can access all the elements in the set through `Iterator iter's` methods `hasNext` and `next`, but the elements are in arbitrary order. The objects returned by the iterator's `next` method are `Map.Entry<K, V>` objects. We show an easier way to do this using the enhanced `for` statement in Example 7.12.

Method `entrySet` and Classes `EntrySet` and `SetIterator`

Method `entrySet` returns a set view of the underlying hash table (its key–value pairs) by returning an instance of inner class `EntrySet`. We define method `entrySet` next and then class `EntrySet`.

```
/** Creates a set view of a map.
    @return a set view of all key-value pairs in this map
 */
public Set<Map.Entry<K, V>> entrySet() {
    return new EntrySet();
}
```

We show the inner class `EntrySet` in Listing 7.11. This class is an extension of the `AbstractSet`, which provides a complete implementation of the `Set` interface except for the `size` and `iterator` methods. The other methods required by the `Set` interface are defined using these methods. Most methods are implemented by using the `Iterator` object that is returned by the `EntrySet.iterator` method to access the contents of the hash table through its set view. You can also use such an `Iterator` object to access the elements of the set view.

LISTING 7.11
The Inner Class `EntrySet`

```
/** Inner class to implement the set view. */
private class EntrySet<K, V> extends AbstractSet<Map.Entry<K, V>> {

    /** Return the size of the set. */
    @Override
    public int size() {
        return numKeys;
    }
```

```
    /** Return an iterator over the set. */
    @Override
    public Iterator<Map.Entry<K, V>> iterator() {
        return new SetIterator<>();
    }
}
```

The final step is to write class `SetIterator`, which implements the `Iterator` interface. The inner class `SetIterator` enables access to the entries in the hash table. The `SetIterator` class implements the `java.util.Iterator` interface and provides methods `hasNext`, `next`, and `remove`. Its implementation is left as a Programming Project (see Project 6).

Classes TreeMap and TreeSet

Besides `HashMap` and `HashSet`, the Java Collections Framework provides classes `TreeMap` and `TreeSet` that implement the `Map` and `Set` interfaces. These classes use a Red–Black tree (Section 9.3), which is a balanced binary search tree. We discussed earlier that the performances for search, retrieval, insertion, and removal operations are better for a hash table than for a binary search tree (expected O(1) versus O(log *n*)). However, the primary advantage of a binary search tree is that it can be traversed in sorted order. Hash tables, however, can't be traversed in any meaningful way. Also, subsets based on a range of key values can be selected using a `TreeMap` but not by using a `HashMap`.

EXAMPLE 7.12 In Example 7.5 we showed how to use a `Map` to build an index for a term paper. Because we want to display the words of the index in alphabetical order, we must store the index in a `TreeMap`. Method `showIndex` below displays the string representation of each index entry in the form

key : value

If the word *fire* appears on lines 4, 8, and 20, the corresponding output line would be

```
fire : [4, 8, 20]
```

It would be relatively easy to display this in the more common form: `fire 4, 8, 20` (see Programming Exercise 4).

```
/** Displays the index, one word per line */
public void showIndex() {
    index.forEach((k, v) -> System.out.println(k + " : " + v));
}
```

The `Map.forEach` method (see Table 7.2) applies a `BiConsumer` (see Table 6.2) to each map key–value pair. In the lambda expression

```
(k, v) -> System.out.println(k + " : " + v)
```

the parameter `k` is bound to the key, and the parameter `v` is bound to the value.

Or the enhanced `for` loop could be used, but the code would be more cumbersome:

```
/** Displays the index, one word per line */
public void showIndex() {
    for (Map.Entry<String, List<Integer>> entry
         : index.entrySet()) {
        System.out.println(entry);
    }
}
```

EXERCISES FOR SECTION 7.5

SELF-CHECK

1. Explain why the nested `interface Map.Entry` is needed.

PROGRAMMING

1. Write statements to display all key–value pairs in `Map` object `m`, one pair per line. You will need to create an iterator to access the map entries.
2. Assume that a `Person` has data fields `lastName` and `firstName`. Write an `equals` method that returns **`true`** if two `Person` objects have the same first and last names. Write a `hashCode` method that satisfies the `hashCode` contract. Make sure that your `hashCode` method does not return the same value for `"Henry James"` and `"James Henry"`. Your `equals` method should return a value of **`false`** for these two people.
3. Assume class `HashSetOpen` is written using an array `table` for storage instead of a `HashMap` object. Write method `contains`.
4. Modify method `showIndex` so each output line displays a word followed by a comma and a list of line numbers separated by commas. You can either edit the string corresponding to each `Map` entry before displaying it or use methods `Map.Entry.getKey` and `Map.Entry.getValue` to build a different string.

7.6 Additional Applications of `Maps`

In this section, we will consider two case studies that use a `Map` object. The first is the design of a contact list for a cell phone, and the second involves completing the Huffman Coding Case Study started in Section 6.7.

CASE STUDY Implementing a Cell Phone Contact list

Problem A cell phone manufacturer would like a Java program that maintains the list of contacts for a cell phone owner. For each contact, a person's name, there should be a list of phone numbers that can be changed. The manufacturer has provided the interface for the software (see Table 7.12).

Analysis A map is an ideal data structure for the contact list. It should associate names (which must be unique) to lists of phone numbers. Therefore, the name should be the key field, and the list of phone numbers should be the value field. A sample entry would be:

name: `Jane Smith` *phone numbers:* `215-555-1234, 610-555-4820`

Thus, we can implement `ContactListInterface` by using a `TreeMap<String, List<String>>` object for the contact list. For the sample entry above, this object would contain the key-value pair `("Jane Smith", ["215-555-1234", "610-555-4820"] )`.

TABLE 7.12
Methods of `ContactListInterface`

Method	Behavior
`List<String> addOrChangeEntry (String name, List<String> numbers)`	Changes the numbers associated with the given `name` or adds a new entry with this `name` and list of numbers. Returns the old list of numbers or **`null`** if this is a new entry
`List<String> lookupEntry(String name)`	Searches the contact list for the given `name` and returns its list of numbers or **`null`** if the `name` is not found
`List<String> removeEntry(String name)`	Removes the entry with the specified `name` from the contact list and returns its list of numbers or **`null`** if the `name` is not in the contact list
`void display()`	Displays the contact list in order by `name`

Design

We need to design the class `MapContactList`, which implements `ContactListInterface`. The contact list can be stored in the data field declared as follows:

```
public class MapContactList implements ContactListInterface {

    /** The contact list */
    Map<String, List<String>> contacts = new TreeMap<>();
```

Writing the required methods using the `Map` methods is a straightforward task.

Implementation

We begin with method `addOrChangeEntry`:

```
public List<String> addOrChangeEntry(String name,
                                      List<String> newNumbers) {
    List<String> oldNumbers = contacts.put(name, newNumbers);
    return oldNumbers;
}
```

Method `put` inserts the new `name` and list of numbers (method arguments) for a `Map` entry and returns the old `value` for that `name` (the list of numbers) if it was previously stored. If an entry with the given `name` was not previously stored, **`null`** is returned.

The `lookupEntry` method uses the `Map.get` method to retrieve the directory entry. The entry key field (`name`) is passed as an argument.

```
public List<String> lookupEntry(String name) {
    return contacts.get(name);
}
```

The `removeEntry` method uses the `Map.remove` method to delete a contact list entry.

```
public List<String> removeEntry(String name) {
    return contacts.remove(name);
}
```

Method `display` uses the `Map.forEach` method with a lambda expression to print each entry in the contact list as explained in Example 7.12.

```
public void display() {
    contacts.forEach((k, v) -> System.out.println(k + '\n' + v));
}
```

The contact list is displayed as a sequence of lines consisting of a name followed by its phone numbers on the next line. The entries are displayed in order by name which is what we desire.

Testing

Below we show three methods for a JUnit testing class. Method `initialize` loads `contactList` with two entries: `{("Jones", ["123", "345"]), ("King", ["135", "357"])}`. The attribute `@BeforeEach` tells JUnit to execute this method before each test. We show two tests for method `lookupEntry`. Programming Exercise 2 asks you to complete the testing class.

```
import org.junit.jupiter.api.BeforeEach;
    private MapContactList contactList = new MapContactList();

    @BeforeEach
    private void initialize() {
              List<String> nums = new ArrayList<>(List.of("123", "345"));
              contactList.addOrChangeEntry("Jones", nums);
              nums = new ArrayList<>(List.of("135", "357"));
              contactList.addOrChangeEntry("King", nums);
    }

    @Test
    public void lookupAnItemNotInList() {
        assertNull(contactList.lookupEntry("Smith"));
    }
    @Test
    public void lookupItemInList() {
        assertEquals(List.of("135", "357"),
                     contactList.lookupEntry("King"));
    }
```

CASE STUDY Completing the Huffman Coding Problem

Problem

In Section 6.7 we showed how to compress a file by using a Huffman tree to encode the symbols occurring in the file so that the most frequently occurring characters had the shortest binary codes. Recall that the custom Huffman tree shown in Figure 6.37 was built for a data file in which the letter e occurs most often, then a and then d. The letters b and c occur least often. The letters are stored in leaf nodes. We used a priority queue to build the Huffman tree.

In this case study, we discuss how to create the frequency table for the symbols in a data file. We use a `Map` for storing each symbol and its corresponding count. We also show how to encode the data file using a second `Map` for storing the code table.

Analysis In Section 6.7, method `buildTree` of class `HuffmanTree` built the Huffman tree. The input to method `buildTree` was a frequency table (array `HuffData`) consisting of (weight, symbol) pairs, where the weight in each pair was the frequency of occurrence of the corresponding symbol in a data file to be encoded.

To encode our data file, we used a code table consisting of (symbol, code) pairs for each symbol in the data file. For both situations, we need to look up a symbol in a table. Using a `Map` ensures that the table lookup is an expected O(1) process. We will also need to write a representation of the Huffman tree to the data file so that we can decode it. We discuss this later.

Design To build the frequency table, we need to read a file and count the number of occurrences of each symbol in the file. The symbol will be the `key` for each entry in `Map<Character, Integer> frequencies`, and the corresponding `value` will be the count of occurrences so far. As each symbol is read, we retrieve its `Map` entry and increment the corresponding count. If the symbol is not yet in the frequency table, we insert it with a count of 1.

The algorithm for building the frequency table follows. After all characters are read, we create a set view of the `Map` and traverse it using an iterator. We retrieve each `Map.Entry` and transpose its fields to create the corresponding `HuffData` array element, a (weight, symbol) pair.

Algorithm for buildFreqTable

1. **`while`** there are more characters in the input file
2. Read a character and retrieve its corresponding entry in `Map frequencies`.
3. Increment `value`.
4. Create a set view of `frequencies`.
5. **`for`** each entry in the set view
6. Store its data as a weight–symbol pair in the `HuffData array`.
7. Return the `HuffData` array.

Once we have the frequency table, we can construct the Huffman tree using a priority queue as explained in Section 6.7. Then we need to build a code table that stores the bit string code associated with each symbol to facilitate encoding the data file. Storing the code table in a `Map<Character, BitString>` object makes the encoding process more efficient because we can look up the symbol and retrieve its bit string code (expected O(1) process). The code table should be declared as:

```
private Map<Character, BitString> codeMap;
```

Method `buildCodeTable` builds the code table by performing a preorder traversal of the Huffman tree. As we traverse the tree, we keep track of the bit code string so far. When we traverse left, we append a 0 to the bit string, and when we traverse right, we append a 1 to the bit string. If we encounter a symbol in a node, we insert that symbol along with a copy of the code so far as a new entry in the code table. Because all symbols are stored in leaf nodes, we return immediately without going deeper in the tree.

Algorithm for Method buildCodeTable

1. Get the data at the current root.
2. **if** a symbol is stored in the current root (reached a leaf node)
3. Insert the symbol and bit string code so far as a new code table entry.
4. **else**
5. Append a 0 to the bit string code so far.
6. Apply the method recursively to the left subtree.
7. Append a 1 to the bit string code.
8. Apply the method recursively to the right subtree.

Finally, to encode the file, we read each character, look up its bit string code in the code table `Map`, and then append it to the output `BitStringBuilder` (described after Listing 7.13).

Algorithm for Method encode

1. **While** there are more characters in the input file
2. Read a character and get its corresponding bit string code.
3. Append its bit string to the output `BitStringBuilder`.

Implementation

Listing 7.12 shows the code for method `buildFreqTable`. The `while` loop inside the `try` block builds the frequency table (`Map frequencies`). Each character is stored as an `int` in `nextChar` and then cast to type `char`. Because a `Map` stores references to objects, each character is autoboxed in a `Character` object (the `key`) and its count in an `Integer` object (the `value`). Once the table is built, we use an enhanced **`for`** loop to traverse the set view, retrieving each entry from the map and using its data to create a new `HuffData` element for array `freqTable`. When we finish, we return `freqTable` as the method result.

LISTING 7.12
Method `buildFreqTable`

```
public static HuffData[] buildFreqTable(BufferedReader ins) {
    // Map of frequencies.
    Map<Character, Integer> frequencies = new HashMap<>();
    try {
        int nextChar; // For storing the next character as an int
        while ((nextChar = ins.read()) != -1) { // Test for more data
            // Get the current count and increment it.
            Integer count = frequencies.getOrDefault((char) nextChar, 0);
            count++;

            // Store updated count.
            frequencies.put((char) nextChar, count);
        }
        ins.close();
    } catch (IOException ex) {
        ex.printStackTrace();
        system.exit(1);
    }

    // Copy Map entries to a HuffData[] array.
    HuffData[] freqTable = new HuffData[frequencies.size()];
    int i = 0;      // Start at beginning of array.
```

```
    // Get each map entry and store it in array freqTable
    // as a weight-symbol pair.
    for (Map.Entry<Character, Integer> entry : frequencies.entrySet()) {
        freqTable[i] =
             new HuffData(entry.getValue(), entry.getKey());
        i++;
    }
    return freqTable;    // Return the array.
}
```

Listing 7.13 shows the method `buildCodeTable`. We provide a starter method that initializes `codeMap` to an empty `HashMap` and calls the recursive method that implements the algorithm discussed in the Design section.

LISTING 7.13
Method `buildCodeTable`

```
/** Starter method to build the code table.
    post: The table is built.
 */
public void buildCodeTable() {
    // Initialize the code map.
    codeMap = new HashMap<>();
    // Call recursive method with empty bit string for code so far.
    buildCodeTable(huffTree, new BitString());
}

/** Recursive method to perform breadth-first traversal of the Huffman tree and
    build the code table.
    @param tree The current tree root
    @param code The code string so far
 */
private void buildCodeTable(BinaryTree<HuffData> tree, BitString code)
{
    // Get data at local root.
    HuffData datum = tree.getData();
    if (datum.symbol != '\u0000') { // Test for a leaf node.
         // Found a symbol, insert its code in the map.
         codeMap.put(datum.symbol, code);
    } else {
         // Append 0 to code so far and traverse left.
         BitString leftCode = code.append(0);
         buildCodeTable(tree.getLeftSubtree(), leftCode);
         // Append 1 to code so far and traverse right
         BitString rightCode = code.append(1);
         buildCodeTable(tree.getRightSubtree(), rightCode);
    }
}
```

The goal of Huffman coding is to produce the smallest possible representation of the text. If we were to write the file as a `String` of 0 and 1 characters, the resulting file would be larger than the original, not smaller! Thus, we need to output a sequence of individual bits, packed into 8-bit bytes. The class `BitString` encapsulates a sequence of bits much like the `String` class encapsulates a sequence of characters. Like the `String`, `BitStrings` are immutable. The `BitString.append` method is equivalent to the `+ operator` in the `String` class in that it creates a new `BitString` with the argument appended to the original. To facilitate

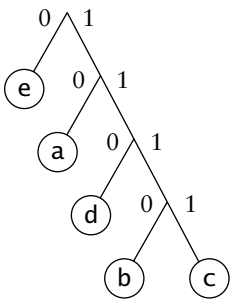
appending individual bits or sequences of bits to a `BitString`, the `BitStringBuilder` provides methods analogous to the `StringBuilder`. Classes `BitString` and `BitStringBuilder` may be downloaded from the Web site for this textbook.

Method `encode` reads each character again, looks up its bit code string, and writes it to the output file. We assume that method `buildCode` has executed and stored the code table in data field `Map<Character, BitString> codeMap`.

Following is the encode method.

```
/** Encodes a data file by writing it in compressed bit string form.
    @param ins The input stream
    @return The coded file as a BitString
 */
public BitString encode(BufferedReader ins) {
    BitStringBuilder result = new BitStringBuilder(); // The complete bit string.
    try {
        int nextChar;
        while ((nextChar = ins.read()) != -1) { // More data?
            // Get bit string corresponding to symbol nextChar.
            BitString nextChunk = codeMap.get((char) nextChar);
            result.append(nextChunk);   // Append to result string.
        }

    } catch (IOException ex) {
        ex.printStackTrace();
        System.exit(1);
    }
    return result.toBitString();
}
```

Testing To test these methods completely, you need to download class `BitString` and `BitStringBuilder` (see Project 1) and write a `main` method that calls them in the proper sequence. For interim testing, you can read a data file and display the frequency table that is constructed to verify that it is correct. You can also use the `StringBuilder` class instead of class `BitString` in methods `buildCodeTable` and `encode`. The resulting code string would consist of a sequence of digit characters `'0'` and `'1'` instead of a sequence of 0 and 1 bits. But this would enable you to verify that the program works correctly.



Encoding the Huffman Tree

Method `decode` in Listing 6.12 used the Huffman tree to decode the encoded file. This tree must be available to the receiver of the encoded file. Therefore, we must encode the Huffman tree and write it to the encoded file.

We can encode the Huffman tree by performing a preorder traversal and appending a 0 bit when we encounter an internal node and a 1 bit when we encounter a leaf. Also, when we reach a leaf, we append the 16-bit Unicode representation of the character. The Huffman tree in Figure 6.37 (reproduced here) is encoded as follows:

```
01e01a01d01b1c
```

where the letters are replaced with their Unicode representations.

Algorithm for encodeHuffmanTree

1. `if` the current root is not a leaf
2. Append a 0 to the output `BitStringBuilder`.
3. Recursively call `encodeHuffmanTree` on the left subtree.
4. Recursively call `encodeHuffmanTree` on the right subtree.

`else`

5. Append a 1 to the output `BitStringBuilder`.
6. Append the 16-bit Unicode of the symbol to the `BitStringBuilder`

Reconstructing the Huffman tree is done recursively. If a 1 bit is encountered, construct a leaf node using the next 16 bits in the `BitString`, and return this node. If a zero bit is encountered, recursively decode the left subtree and the right subtree, then create an internal node with the resulting subtrees.

EXERCISES FOR SECTION 7.6

SELF-CHECK

1. If `contactList` is type `MapContactList` show the result returned by each statement below or the effect of its execution if the result is `void`. Assume they are executed in sequence.
   a. `contactList.lookupEntryeEntry("Adam");`
   b. `contactList.addOrChangeEntry("Adam", List.of("611"));`
   c. `contactList.addOrChangeEntry("Bob", List.of("800"));`
   d. `contactList.display();`
   e. `contactList.removeEntry("Sam");`
   f. `contactList.removeEntry("Adam");`
   g. `contactList.removeEntry("Adam");`

PROGRAMMING

1. Write a JUnit testing class to complete the test of the `MapContactList` class.

7.7 Navigable Sets and Maps

The `SortedSet` interface extends the `Set` by enabling the user to get an ordered view of the elements with the ordering defined by a `compareTo` method or by means of a `Comparator`. Because the items have an ordering, additional methods can be defined, which return the first and last elements and define subsets over a specified range. The `SortedMap` interface provides an ordered view of a map with the elements ordered by key value. Because the elements of a submap are ordered, submaps can also be defined.

The newer `NavigableSet` and `NavigableMap` interfaces were introduced as an extension to `SortedSet` and `SortedMap`. These interfaces allow the user to specify whether the start or end items are included or excluded. They also enable the user to specify a subset or submap that is traversable in the reverse order. As they are more general, we will discuss the `NavigableSet` and `NavigableMap` interfaces. Java retains the `SortedSet` and `SortedMap` interfaces for compatibility with existing software. Table 7.13 shows some methods of the `NavigableSet` interface.

TABLE 7.13
`NavigableSet` Interface

Method	Behavior
`E ceiling(E e)`	Returns the smallest element in this set that is greater than or equal to `e`, or `null` if there is no such element
`Iterator<E> descendingIterator()`	Returns an `iterator` that traverses the `Set` in descending order
`NavigableSet<E> descendingSet()`	Returns a reverse order view of this set
`E first()`	Returns the smallest element in the set
`E floor(E e)`	Returns the largest element that is less than or equal to `e`, or `null` if there is no such element
`NavigableSet<E> headset(E toEl, boolean incl)`	Returns a view of the subset of this set whose elements are less than `toEl`. If `incl` is **`true`**, the subset includes the element `toEl` if it exists
`E higher(E e)`	Returns the smallest element in this set that is strictly greater than `e`, or `null` if there is no such element
`Iterator<E> iterator()`	Returns an `iterator` to the elements in the set that traverses the set in ascending order
`E last()`	Returns the largest element in the set
`E lower(E e)`	Returns the largest element in this set that is strictly less than `e`, or `null` if there is no such element
`E pollFirst()`	Retrieves and removes the first element. If the set is empty, returns `null`
`E pollLast()`	Retrieves and removes the last element. If the set is empty, returns `null`
`NavigableSet<E> subSet(E fromEl, boolean fromIncl, E toEl, boolean toIncl)`	Returns a view of the subset of this set that ranges from `fromEl` to `toEl`. If the corresponding `fromIncl` or `toIncl` is **`true`**, then the `fromEl` or `toEl` elements are included
`NavigableSet<E> tailSet(E fromEl, boolean incl)`	Returns a view of the subset of this set whose elements are greater than `fromEl`. If `incl` is **`true`**, the subset includes the element `fromE` if it exists

EXAMPLE 7.13 Listing 7.14 illustrates the use of a `NavigableSet`. The output of this program consists of the lines:

```
The ordered set odds is [1, 3, 5, 7, 9]
The ordered set b is [3, 5, 7]
Its first element is 3
Its smallest element >= 6 is 7
```

LISTING 7.14
Using a `NavigableSet`

```
public static void main(String[] args) {
    // Create and fill the sets
    NavigableSet<Integer> odds = new TreeSet<>(Set.of(5, 3, 7, 1, 9));
    System.out.println("The original set odds is " + odds);
    NavigableSet b = odds.subSet(1, false, 7, true);
    System.out.println("The ordered set b is " + b);
    System.out.println("Its first element is " + b.first());
    System.out.println("Its smallest element >= 6 is " + b.ceiling(6));
}
```

The methods defined by the NavigableMap interface are similar to those defined in NavigableSet except that the parameters are keys and the results are keys, Map.Entrys, or submaps. For example, the NavigableSet has a method ceiling that returns a single set element or **null** while the NavigableMap has two similar methods: ceilingEntry(K key), which returns the Map.Entry that is associated with the smallest key greater than or equal to the given key, and ceilingKey(K key), which returns just the key of that entry.

Table 7.14 shows some methods of the NavigableMap interface. Not shown are methods firstKey, firstEntry, floorKey, floorEntry, lowerKey, lowerEntry, lastKey, lastEntry, higherKey, and higherEntry, which have the same relationship to their NavigableSet counterparts (methods first, floor, etc.) as do ceilingKey and ceilingEntry.

The entries in a NavigableMap can be processed in key–value order. This is sometimes a desirable feature, which is not available in a HashMap (or hash table). Class TreeMap and class ConcurrentSkipListMap implement the NavigableMap interface. To use the class ConcurrentSkipListSet or ConcurrentSkipListMap, you must insert both of the following statements.

```
import java.util.*;
import java.util.concurrent.*;
```

Application of a NavigableMap

Listing 7.15 shows methods computeAverage and computeSpans. The method computeAverage computes the average of the values defined in a Map. Method computeSpans creates a group of submaps of a NavigableMap and passes each submap to computeAverage. Given a NavigableMap in which the keys represent years and the values some statistic for that year, we can generate a table of averages covering different periods. For example, if we have a

TABLE 7.14
NavigableMap Interface

Method	Behavior
Map.Entry<K, V> ceilingEntry(K key)	Returns a key–value mapping associated with the least key greater than or equal to the given key, or **null** if there is no such key
K ceilingKey(K key)	Returns the least key greater than or equal to the given key, or **null** if there is no such key
NavigableSet<K> descendingKeySet()	Returns a reverse-order NavigableSet view of the keys contained in this map
NavigableMap<K, V> descendingMap()	Returns a reverse-order view of this map
NavigableMap<K, V> headMap(K toKey, boolean incl)	Returns a view of the submap of this map whose keys are less than toKey. If incl is **true**, the submap includes the entry with key toKey if it exists
NavigableMap<K, V> subMap(K fromKey, boolean fromIncl, K toKey, boolean toIncl)	Returns a view of the submap of this map that ranges from fromKey to toKey. If the corresponding fromIncl or toIncl is **true**, then the entries with key fromKey or toKey are included
NavigableSet<E> tailMap(K fromKey, boolean fromIncl)	Returns a view of the submap of this map whose elements are greater than fromKey. If fromIncl is **true**, the submap includes the entry with key fromKey if it exists
NavigableSet<K> navigableKeySet()	Returns a NavigableSet view of the keys contained in this map

map of storms that represents the number of tropical storms in a given year for the period 1960–1969, the method call

```
List<Number> stormAverage = computeSpans(storms, 2);
```

will calculate the average number of storms for the years 1960–1961, 1962–1963, and so on and store these values in the `List stormAverage`. The value passed to `delta` is 2, so within method `computeSpans`, the first statement below

```
double average =
    computeAverage(valueMap.subMap(index, true,
                                   index+delta, false));
result.add(average);
```

will create a submap for a pair of years (years `index` and `index+1`) and then compute the average of the two values in this submap. The second statement appends the average to the `List result`. This would be repeated for each pair of years starting with the first pair (1960–1961). For the method call

```
List<Number> stormAverage = computeSpans(storms, 3);
```

a submap would be created for each non-overlapping group of 3 years: 1960–1962, 1963–1965, 1966–1968, and then 1969 by itself. The average of the three values in each submap (except for the last, which contains just one entry) would be calculated and stored in `List result`.

LISTING 7.15

Methods `computeAverage` and `computeSpans`

```
/** Returns the average of the numbers in its Map argument.
    @param valueMap The map whose values are averaged
    @return The average of the map values
 */
public static double computeAverage(Map<Integer, Double> valueMap) {
    int count = 0;
    double sum = 0;
    for (Map.Entry<Integer, Double> entry : valueMap.entrySet()) {
        sum += entry.getValue().doubleValue();
        count++;
    }
    return (double) sum / count;
}

/** Returns a list of the averages of nonoverlapping spans of
    values in its NavigableMap argument.
    @param valueMap The map whose values are averaged
    @param delta The number of map values in each span
    @return An ArrayList of average values for each span
 */
public static List<Double> computeSpans(NavigableMap valueMap, int delta)
{

    List<Double> result = new ArrayList<>();
    Integer min = (Integer) valueMap.firstEntry().getKey();
    Integer max = (Integer) valueMap.lastEntry().getKey();
    for (int index = min; index <= max; index += delta) {
        double average =
            computeAverage(valueMap.subMap(index, true,
                                           index+delta, false));
        result.add(average);
    }
    return result;
}
```

EXERCISES FOR SECTION 7.7

SELF-CHECK

1. What is displayed by the execution of the following program?

```
public static void main(String[] args) {
    NavigableSet<Integer> s = new TreeSet<>(Set.of(5, 6, 3, 2, 9));
    NavigableSet<Integer> a = s.subSet(1, true, 9, false);
    NavigableSet<Integer> b = s.subSet(4, true, 9, true);
    System.out.println(a);
    System.out.println(b);
    System.out.println(s.higher(5));
    System.out.println(s.lower(5));
    System.out.println(a.first0);
    System.out.println(b.lower(4));
    int sum = 0;
    for (int i : a) {
      System.out.println(i);
      sum += i;
    }
    System.out.println(sum);
}
```

2. Trace the execution of methods `computeSpans` and `computeAverage` for the call `computeSpans (storms, 4)` where `NavigableMap<Integer, Double> storms` has the following entries: {(1960, 10), (1961, 5), (1962, 20), (1963, 8), (1964, 16), (1965, 50), (1966, 25), (1967, 15), (1968, 21), (1969, 13)}.

3. Write an algorithm for method `computeGaps` that has two parameters like `computeSpans`, except the `int` parameter represents the number of years between entries in the `NavigableMap` that are being averaged together. For `NavigableMap` storms defined in the previous exercise, the call `computeGaps(storms, 2)` would first compute the average for the values in {(1960, 10), (1962, 20), (1964, 16) . . . } and then compute the average for the values in {(1961, 5), (1963, 8), (1965, 50) . . . }.

PROGRAMMING

1. Write a program fragment to display the elements of a set `s` in normal order and then in reverse order.

2. Write a main method that tests method `computeSpans`.

3. Code method `computeGaps` from Self-Check Exercise 3.

7.8 Skip-Lists

The skip-list was invented by William Pugh in 1989. The elements of a skip-list are kept sorted according to their natural ordering or by a `comparator`. A skip-list can be thought of as a linked list whose nodes are linked in multiple ways. The Java `ConcurrentSkipListSet` and `ConcurrentSkipListMap` implement the `NavigableSet` and `NavigableMap` interface, respectively, and support concurrency.

The concurrency features are beyond the scope of this text, but we will describe the basic structure of the skip-list. Our skip-list will implement some of the methods of the `KWHashMap`

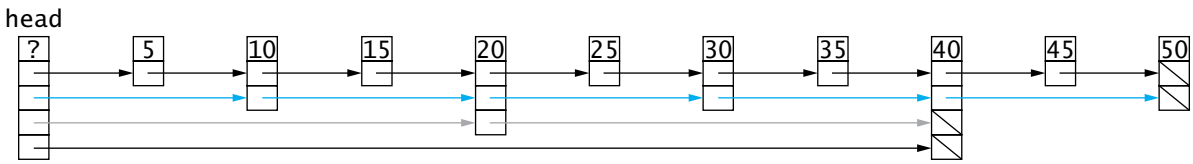
interface described in Table 7.3. We will focus on the algorithms for search (`find`), retrieval (`get`), and insertion (`put`). We will show that these algorithms have O(log *n*) performance.

Skip-List Structure

A skip-list is a list of lists. Each node in a list contains an `Entry` (see Listing 7.3) that contains a key–value pair. The elements in each list are in increasing order by key. Unlike the usual list node, which has a single forward link to the next node, the nodes in a skip-list have a varying number of forward links. The number of such links is determined by the level of a node. A level *m* node has *m* forward links. When a new data element is inserted in a skip-list, a new node is inserted to store the element. The node's level is chosen randomly in such a way that approximately 50 percent are level 1 (one forward link), 25 percent are level 2 (two forward links), 12.5 percent are level 3, and so on.

Figure 7.8 shows a skip-list with 10 data elements. To simplify the discussion, we are showing only the keys (`int`s in this case) because they determine the ordering of nodes in the skip-list and the structure of the skip-list. In the actual skip-list, each key would be replaced by a reference to an `Entry` item. In this skip-list, there are five level 1 nodes (5, 15, 25, 35, and 45), three level 2 nodes (10, 30, and 50), one level 3 node (20), and one level 4 node (40). If we look at node 20, we see that its level 1 link (the top one) is to node 25, its level 2 link is to node 30, and its level 3 link (the bottom one) is to node 40. The last node in the skip-list, node 50, is a level 2 node, and both links are **`null`**.

FIGURE 7.8
Ideal Skip-List



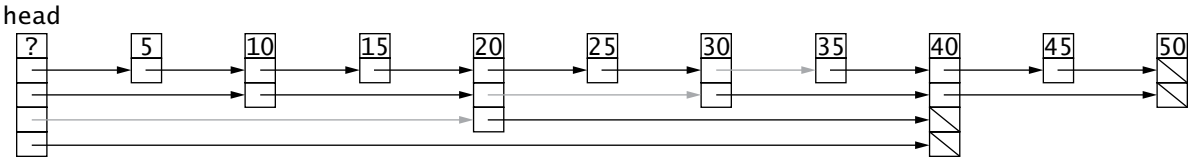
The *level* of a skip-list is defined as its highest node level, or 4 for Figure 7.8. A level 4 skip-list has individual lists of level 4, level 3, level 2, and level 1. The level 1 list consists of every node (5, 10, 15, etc.); the level 2 list (in color) consists of every other node (10, 20, 30, 40, 50); the level 3 list (in gray) consists of nodes 20 and 40; and the level 4 list consists of node 40. This is an *ideal skip-list*; most skip-lists will not have this exact structure, but their behavior will be similar.

Searching a Skip-List

To search a skip-list, we start by looking for our target in the highest-level list. This list always has the fewest elements. If the target is in this list, the search is successful. If not, we stop the search in the current list at the element that is the predecessor of the target. Then we continue the search in the list with level one less than the current list, starting where we left off (the predecessor to the target). We continue this process until we either find the target or reach the level 1 list. If the target is not in the level 1 list (the list of all elements), then it is not present.

Figure 7.9 shows the search path for item 35. We start by searching the level 4 list. Its first node is 40 (>35), so we move to the level 3 list. Its first node is 20 and its second node is 40

FIGURE 7.9
Searching for 35 in a Skip-List



(>35), so we stop at node 20. We then follow node 20's level 2 link, which points to a node whose value is 30. Advancing to the 30-node, we see that its level 2 link is to 40 (>35), so we stop at node 30. We then follow its level 1 link, which points to 35, our target. The gray links point to the predecessor of 35 in each list.

The algorithm for searching a skip-list follows:

1. Let *m* be the highest-level node.
2. **while** *m* > 0
3. Following the level *m* links, find the node with the largest key value that is less than or equal to the target.
4. **if** it is equal to the target, the target has been found—exit loop.
5. Set *m* to *m* – 1
6. **if** *m* is 0, the target is not in the list.

Performance of a Skip-List Search

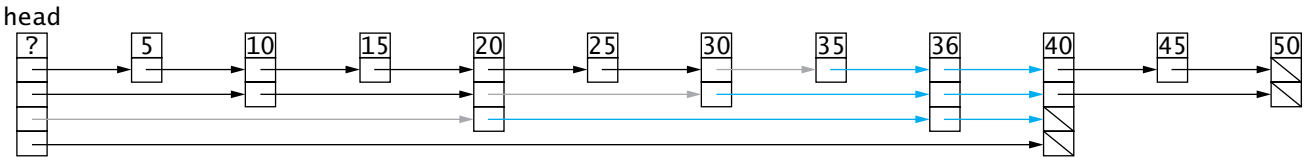
Because the first list we search has the fewest elements (generally one or two) and each lower-level list we search has approximately half as many elements as the current list, the search performance is similar to that of a binary search: O(log *n*), where *n* is the number of nodes in the skip-list. We discuss performance further at the end of this section.

Inserting into a Skip-List

If the search algorithm fails to find the target, it will find its predecessor in the level 1 list, which is the target's insertion point. Therefore, if we keep track of the last node we visited at each level, we know where to insert a new node containing the target. The question then is "What level should this new node be?" The answer is it is chosen at random, based on the number of items currently in the skip-list. The random number is chosen with a logarithmic distribution. Half the time a level 1 node is chosen; a quarter of the time a level 2 node is chosen, and $1/2^m$ time a level *m* node is chosen.

To insert 36 into the skip-list shown in Figure 7.9, we would follow the same path as to locate 35. Along the way we would have recorded the last node visited at each level: the 20 node at level 3, the 30 node at level 2, and the 35 node at level 1. If the random number generator returns a 3, the new node will be a level 3 node. The level 3 link in the 20 node will be set to point to the new 36 node, and the level 3 link in the new 36 node will be set to point to node 40. The level 2 link in the 30 node will be set to point to the new 36 node, and the level 2 link in the new 36 node will be set to point to node 40. Finally, the level 1 link in the 35 node will be set to point to the new 36 node, and the level 1 link in the new 36 node will be set to point to the 40 node. The result is shown in Figure 7.10. The color links show the new entries in the skip-list.

FIGURE 7.10
After Insertion of 36



Increasing the Height of a Skip-List

The skip-list shown in the figures is a level 4 skip-list. Such a skip-list effectively can hold up to 15 items. When a 16th item is inserted, the level is increased by 1. A level m skip-list can effectively hold up to $2^m - 1$ items. If more items were inserted without increasing the level, then the O(log n) performance would not be achieved.

Implementing a Skip-List

Next, we show how to implement a skip-list. We start with the `SLNode` class (Listing 7.16). When a new level m node is created, data field `item` stores its `key` and `value`. Data field `links` allocates array links with subscripts 0 through $m - 1$. We use the same kind of node for node `head` as the rest of the nodes in the skip-list. However, its `key` and `value` fields are **`null`** (indicated by `?` in the figures).

LISTING 7.16
`SLNode` class

```
/** Static class to contain the data and the links in a skip-list
    @param <K> The key type
    @param <V> The value type
 */
private class SLNode<K, V> {
    private SLNode<K, V>[] links;
    private Entry<K, V> item;
    // Create a node of level m
    public SLNode(int m, K key, V value) {
        item = new Entry<>(key, value);
        links = (SLNode<K, V>[]) new SLNode[m]; // create links
    }

 }
```

Next we provide the imports, heading, and declarations for class `SkipList` and a constructor in Listing 7.17.

LISTING 7.17
`SkipList` Data Field Declarations and Constructor

```
import java.util.Arrays;
import java.util.Random;
/** Class to implement a skip-list
    @param K type of key
    @param V type of value
 */
```

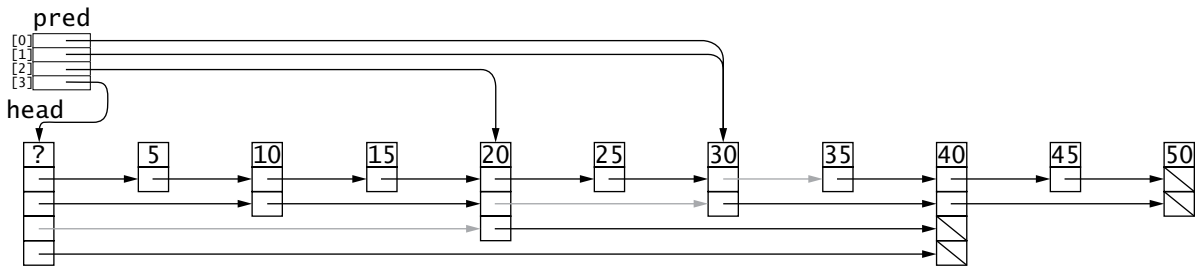
```
public class SkipList<K extends Comparable<K>, V> {
    // Insert class Entry (Listing 7.6) here.
    // Insert class SLNode (Listing 7.3) here.
    /** The current size of the SkipList */
    private int size;
    /** Maximum level */
    private int maxLevel;
    /** Nominal maximum capacity */
    private int maxCap;
    /** The head node is a dummy node, it contains no data */
    private final SLNode<K, V> head;
    /** Construct an initially empty SkipList. */
    public SkipList() {
        size = 0;
        maxLevel = 1;
        maxCap = 1;
        head = new SLNode<K, V>(maxLevel, null, null);
    }
```

SkipList Methods for Search and Retrieval

Since insertion involves the same algorithm as searching to find the insertion point, we define a common method, `search`, which will return an array `pred` of references to the `SLNodes` at each level that were last examined in the search. Because array subscripts start at 0, `pred[i]` references the predecessor in the level (`i+1`) list of the target. The level (`i+1`) link for the node referenced by `pred[i]` (`pred[i].link[i]`) references a node whose `key` is greater than or equal to `target`. It will be **`null`** if `target` is greater than all items in that sub-list.

The result of searching for 35 is shown in Figure 7.11. Array element `pred[3]` references node `head` since the level 4 link references the node 40, which is greater than 35. Array element `pred[2]` references node 20 whose level 3 link references node 40. Element `pred[1]` references node 30 whose level 2 link references 40 and whose level 1 link references 35. Finally, `pred[0]` also references node 30. By examining `links[0]` of the node referenced by `pred[0]`, we can determine whether `target` is in the skip-list.

FIGURE 7.11
Result of Search for 35



Listing 7.18 shows the code for searching a skip-list. Two methods are shown: `search` and `get`. Method `get` calls `search` to perform the search. The result returned by `search` is the array of references to the predecessor of `target` in each list. The `SkipList` data field `head` references an array of links to the first element of each list in the skip-list, where `head.links[i]` references the first node in the level (`i+1`) list.

Method `search` begins by setting `current` to `head`. The `for` loop ensures that each list is processed beginning with the highest-level list. The `while` loop advances `current` down the level `i`

list until current references the last node in the list or current references a node that is linked to either target or to the first element greater than target. The last value of current is saved in pred[i], and the for loop sets i to i-1, causing the next list to be searched.

After search returns the array of predecessor references, method get examines the level 1 link of the predecessor saved for the level *1* list. If this link is **null** or if it references a node greater than target, **null** is returned (target is not in the list); otherwise, the data stored in the node referenced by the level 1 link is returned as the retrieval result.

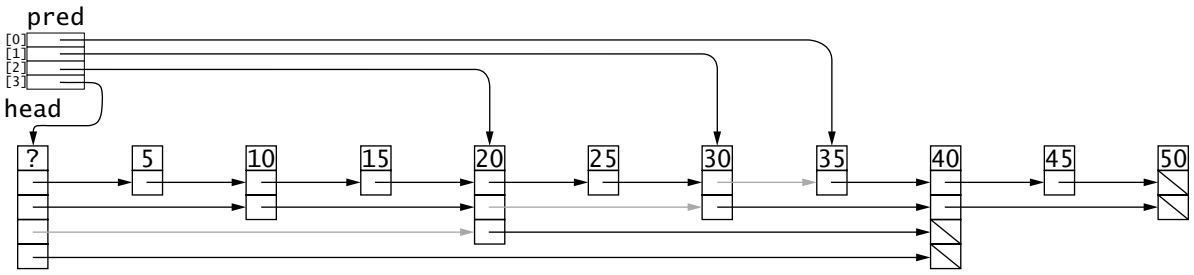
LISTING 7.18
Methods for Searching a Skip-List

```
    @SuppressWarnings("unchecked")
    /** Search for an item in the list
        @param target The key being sought
        @return A SLNode array which references the predecessors
                of the target at each level.
     */
    private SLNode<K, V>[] search (K target) {
        var pred = (SLNode<K, V>[]) new SLNode[maxLevel];
        var current = head;
        for (int i = current.links.length-1; i >= 0; i--) {
           while (current.links[i] != null
             && current.links[i].item.key.compareTo(target) < 0) {
              current = current.links[i];
           }
           pred[i] = current;
        }
        return pred;
    }


/** Retrieve the value field of the object in the skip-list with
    key matching target.
    @param target The key of the item being sought
    @return The old value of the object in the skip-list whose
            key matches target. If not found, return null.
 */

public V get(K target) {
    // Call search to get array of predecessors
    var pred = search(targ);
    if (pred[0].links[0] != null &&
            pred[0].links[0].item.key.compareTo(target) == 0) {
        return pred[0].links[0].item.value;
    } else {
        return null;
    }
}
```

Method put for Inserting into a Skip-List

If we want to insert an item in a skip-list, we must first determine if it is already in the list—duplicate keys are not allowed—and if it is not in the skip-list, we must determine where it should be placed. If it is in the list, we will update its value. As discussed earlier, method search does most of the work for us by advancing pred[0].links[0] to the item we are attempting to insert, or to its successor if it is not in the list. Also, pred[1].links[1] points to its successor in the level 2 list, and so on. Figure 7.12 shows the result of calling search for 36.

FIGURE 7.12
Result of Search for 36



Method `put` begins by calling `search`, passing its argument (`target`) to `search`. Method `search` advances the elements of array `pred` as described earlier and returns array `pred`. If `pred[0].links[0].item.key` matches `target`, method `put` updates the `value` field of `item` and returns its previous `value` using method `Entry.setValue` (see Listing 7.3). Otherwise `put` must splice the new `Entry` into the skip-list and return **`null`**.

Method `put` begins with the following code. The details of inserting a new node are covered in the code fragments that follow.

```
/** Inserts an item in the skip-list - if the key
    is in the skip-list, change its value
    @param target The key being sought
    @param value The value for the key
    @return old value if key found; else, return null
*/
public V put(K target, V value) {
    // Call search to get array of predecessors
    var pred = search(target);

    // Check the item reached in the level 1 list
    if (pred[0].links[0] != null
         && pred[0].links[0].items.key.compareTo(target) == 0) {
         // target matches a key in the list
         // update its value and return the old value
         return pred[0].links[0].data.setValue(value);
    } else {
         // The target was not found, insert new item into skip-list
         // Insertion code goes here . . .
```

Before inserting the new item, the value of `size` is incremented. Recall that a skip-list of level m has a maximum capacity (`maxCap`) of $2^m - 1$ (returned by method `computeMaxCap`—not shown). If `size` is greater than the current `maxCap`, the skip-list is full and the level of the skip-list and its maximum capacity need to be increased. The arrays `head.links` and `pred` are expanded using the `Arrays.copyOf` method and the last element of `pred` is set to reference `head`. Then we set the level of the new node to the new skip-list level. If the skip-list is not full, the level of the new node is determined randomly by method `logRandom`.

```
// Create a new node and splice it into the skip-list
SLNode<K, V> newNode;    // new node to be inserted
int levelNewNode;        // level of new node
// Increment size and check for full skip-list
size++;
if (size > maxCap) {
    /** Skip-list is full - update skip-list data fields,
        set level of new node to new maxLevel */
```

```
                maxLevel++;
                maxCap = computeMaxCap(maxLevel);
                head.links = Arrays.copyOf(head.links, maxLevel);
                pred = Arrays.copyOf(pred, maxLevel);
                pred[maxLevel - 1] = head;
                levelNewNode = maxLevel;
            } else
                levelNewNode = logRandom(); // Set level randomly
            }
```

Next, we store the new node's data and splice it into the skip-list. Figure 7.12 shows that for each level (i + 1), the new node should be inserted between the node referenced by `pred[i]` and its successor, `pred[i].links[i]`. The `for` statement accomplishes this by setting `newNode.links[i]` to reference `pred[i].links[i]` and then resetting `pred[i].links[i]` to reference `newNode`.

```
            // Store data in newNode and splice it into the skip-list
            newNode = new SLNode<E>(levelNewNode, target, value);
            for (int i = 0; i < levelNewNode; i++) {
                newNode.links[i] = pred[i].links[i];
                pred[i].links[i] = newNode;
            }
            return null;  // new node inserted
        } // end else
    } // end put
```

Constants and Methods for Computing Random Level

We want the size of the nodes to be distributed logarithmically. The random number class, `Random`, has a method `nextInt(int n)`, which returns a uniformly distributed random integer from 0 up to, but not including, n. If we compute the $\log_2$`maxCap` + 1, we get a logarithmically distributed random number between 1 and `maxLevel` (the skip-list level). This number has the opposite distribution of what we desire: 1/2 of the numbers will be `maxLevel-1`, 1/4 of the numbers will be `maxLevel-2`, and so on. Thus, we subtract the result from `maxLevel`. Method `logRandom` performs the computation.

```
/** RAND is an instance of a Random number generator */
private static final Random RAND = new Random();

/** LOG2 is the natural Log of 2 */
private static final double LOG2 = Math.log(2.0);

/** Method to generate a logarithmic distributed integer
    between 1 and maxLevel. i.e., 1/2 of the values returned
    are  1, 1/4 are 2, 1/8 are 3, etc.
    @return a random logarithmic distributed int between 1 and maxLevel
 */
private int logRandom() {
    int r = RAND.nextInt(maxCap);
    int k = (int) (Math.log(r + 1) / LOG2);
    if (k > maxLevel - 1) k = maxLevel - 1;
    return maxLevel - k;
}
```

Performance of a Skip-List

In an ideal skip-list (see Figure 7.10), every other node is at level 2, and every 2^{m+1}th node is at least level m+1. With this ideal structure, searching is the same as a binary search;

each repetition reduces the search population by 1/2, and thus the search is O(log *n*). By randomly choosing the levels of inserted nodes to have an exponential distribution, the skip-list will have the desired distribution of nodes. However, they will be randomly positioned through the skip-list. Therefore, on the average, the time for search and insertion will be O(log *n*).

Testing Class Skip-List

While testing, it would be useful to provide a `size` and `toString` method (see Programming Exercise 2). Some of the tests that you need to make for methods `put` and `get` follow. Programming Exercise 3 asks you to provide a JUnit testing class to perform the tests below.

- Test that method `put` inserts an item into an empty skip-list.
- Test that method `put` inserts a new item in a skip-list whose key is smaller than the first item in the skip-list.
- Test that method `put` inserts a new item whose key is larger than the largest item in the skip-list.
- Test that method `put` inserts a new item whose key is not the smallest or largest in the skip-list.
- Test that method `put` replaces the value of an item if its key is already in the skip-list and that `put` returns the old value.
- Test that method `get` returns the value in the skip-list if the target key is in the skip-list.
- Test that method `get` returns **`null`** if the skip-list is empty,
- Test that method `get` returns **`null`** if the target key is not found,
- Test that method `get` returns the item if it is smaller than the first item in the skip-list or larger than the largest item in the skip-list.
- Test that method `get` returns the item if it is the first or last item in the skip-list.

EXERCISES FOR SECTION 7.8

SELF-CHECK

1. Show the skip-list after inserting the values 11, 12, 22, and 33 into the skip-list shown in Figure 7.10. Assume that the random number generator returned 2, 1, 3, and 1 for the new node levels.
2. Draw the ideal skip-list for storing the numbers 5, 10, 15, 20, 25, 30, 36, 42, 45, 50, 55, 60, 68, 72, 86, 93.

PROGRAMMING

1. Write method `toString` for a skip-list with integer keys and string values. Method `SkipList.toString()` should call the `toString` methods for classes `Entry` and `SLNode`.
2. Write a JUnit testing class to thoroughly test methods `get` and `put`.
3. Write method `SkipList.remove`. To remove a node from a skip-list, you need to change all references to the node being deleted to reference its successors.

Chapter Review

- The `Set` interface describes an abstract data type that supports the same operations as a mathematical set. We use `Set` objects to store a collection of elements that are not ordered by position. Each element in the collection is unique. Sets are useful for determining whether a particular element is in the collection, not its position or relative order.
- We showed how to use the factory method `Set.of` to create an immutable `Set` and `Map.of` to create an immutable `Map`. We also showed how to use a copy constructor to create a modifiable `Set` or `Map` with the same contents.
- The `Map` interface describes an abstract data type that enables a user to access information (a value) corresponding to a specified key. Each key is unique and is mapped to a value that may or may not be unique. Maps are useful for retrieving or updating the value corresponding to a given key.
- A hash table uses hashing to transform an item's key into a table index so that insertions, retrievals, and deletions can be performed in expected O(1) time. When the `hashCode` method is applied to a key, it should return an integer value that appears to be a random number. A good `hashCode` method should be easy to compute and should distribute its values evenly throughout the range of `int` values. We use modulo division to transform the hash code value to a table index. Best performance occurs when the table size is a prime number.
- A collision occurs when two keys hash to the same table index. Collisions are expected, and hash tables utilize either open addressing or chaining to resolve collisions. In open addressing, each table element references a key–value pair, or **`null`** if it is empty. During insertion, a new entry is stored at the table element corresponding to its hash index if it is empty; otherwise, it is stored in the next empty location following the one selected by its hash index. In chaining, each table element references a linked list of key–value pairs with that hash index or **`null`** if none does. During insertion, a new entry is stored in the linked list of key–value pairs for its hash index.
- In open addressing, linear probing is often used to resolve collisions. In linear probing, finding a target or an empty table location involves incrementing the table index by 1 after each probe. This approach may cause clusters of keys to occur in the table, leading to overlapping search chains and poor performance. To minimize the harmful effect of clustering, quadratic probing increments the index by the square of the probe number. Quadratic probing can, however, cause a table to appear to be full when there is still space available, and it can lead to an infinite loop.
- The best way to avoid collisions is to keep the table load factor relatively low by rehashing when the load factor reaches a value such as 0.75 (75 percent full). To rehash, you increase the table size and reinsert each table element.
- In open addressing, you can't remove an element from the table when you delete it, but you must mark it as deleted. In chaining, you can remove a table element when you delete it. In either case, traversal of a hash table visits its entries in an arbitrary order.
- A set view of a `Map` can be obtained through method `entrySet`. You can create an `Iterator` object for this set view and use it to access the elements in the set view in order by key value.
- Two Java API implementations of the `Map` (`Set`) interface are `HashMap` (`HashSet`) and `TreeMap` (`TreeSet`). The `HashMap` (and `HashSet`) implementation uses an underlying hash table; the `TreeMap` (and `TreeSet`) implementations use a Red–Black tree (discussed in

Chapter 9). Search and retrieval operations are more efficient using the underlying hash table (expected O(1) versus O(log *n*)). The tree implementation, however, enables you to traverse the key–value pairs in a meaningful way and allows for subsets based on a range of key values.

- The `NavigableSet` and `NavigableMap` interfaces enable the creation of subsets and submaps that may or may not include specified boundary items. The elements in a `NavigableSet` (or `NavigableMap`) are in increasing order by element value (or by key value for a `NavigableMap`).
- Java uses a skip-list to implement the `NavigableMap` and `NavigableSet` interfaces. The links in each node of a skip-list form sub-lists for different levels of the skip-list where the size of each sub-list is inversely proportional to its level in a perfect skip-list. A search for a key begins with the smallest sub-list and if unsuccessful it continues with the next larger sub-list. A reference to the node containing the target key or to its predecessor at each level is saved in an array of links to facilitate searching for a key and inserting and retrieving items in the skip-list.

Java API Interfaces and Classes Introduced in This Chapter

```
java.util.AbstractMap
java.util.AbstractSet
java.util.concurrent.ConcurrentSkipListMap
java.util.concurrent.ConcurrentSkipListSet
java.util.HashMap
java.util.HashSet
java.util.Map
java.util.Map.Entry
java.util.NavigableMap
java.util.NavigableSet
java.util.Set
java.util.TreeMap
java.util.TreeSet
```

User-Defined Interfaces and Classes in This Chapter

```
BitString                HashtableChain
ContactListInterface     HashtableOpen
Entry                    KWHashMap
EntrySet                 MapContactList
HashSetOpen              SetIterator
SkipList                 SLNode
```

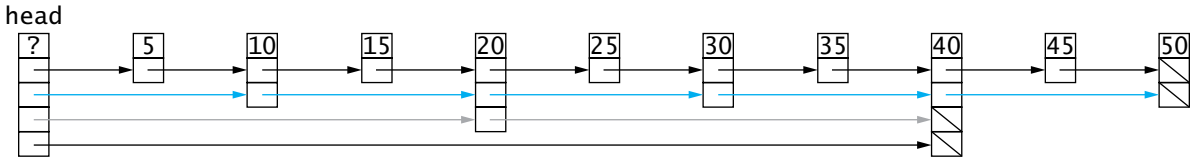
Quick-Check Exercises

1. Write a statement to create a set and load it with `'a'`, `'b'`, `'c'`. Write a statement to insert the character `'d'`.
2. What is the effect of each of the following method calls, given the set in Exercise 1, and what does it return?

```
s.add('a');
s.add('A');
next = 'b';
s.contains(next);
```

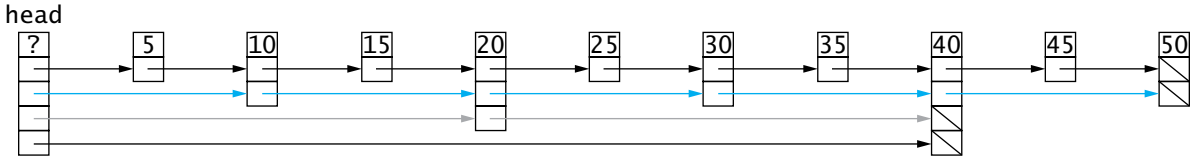
For Questions 3–7, a `Map` `m`, contains the following entries: `(1234, "Jane Doe")`, `(1999, "John Smith")`, `(1250, "Ace Ventura")`, `(2000, "Bill Smythe")`, `(2999, "Nomar Garciaparra")`.

3. What is the effect of the statement `m.put(1234, "Jane Smith");`? What is returned?
4. What is returned by `m.get(1234)`? What is returned by `m.get(1500)`?
5. If the entries for `Map m` are stored in a hash table of size 1000 with open addressing and linear probing, where would each of the items be stored?
6. Answer Question 5 for the case where the entries were stored using quadratic probing.
7. Answer Question 5 for the case where the entries were stored using chaining.
8. What class does the Java API provide that facilitates coding an implementer of the `Map` interface? Of the `Set` interface?
9. List two classes that the Java API provides that implement the `Map` interface. List two that implement the `Set` interface.
10. You apply method _____ to a `Map` to create a set view. You apply method _____ to this set view to get an object that facilitates sequential access to the Map elements.
11. For the following skip-list, what nodes would be referenced by the elements of array `pred` after a search for 17?



Review Questions

1. Show where the following keys would be placed in a hash table of size 5 using open addressing: 1000, 1002, 1007, 1003. Where would these keys be after rehashing to a table of size 11?
2. Answer Question 1 for a hash table that uses chaining.
3. Write a `toString` method for class `HashtableOpen`. This method should display each table element that is not **`null`** and is not deleted.
4. Class `HashtableChain` uses the class `LinkedList`, which is implemented as a double-linked list. Write the `put` method using a single-linked list to hold elements that hash to the same index.
5. Write the `get` method for the class in Question 4.
6. Write the `remove` method for the class in Question 4.
7. Write inner class `EntrySet` for the class in Question 4 (see Listing. 7.11).
8. For the skip-list below what nodes would be referenced by the elements of array `pred` after a search for 42? If you are inserting 42 in the skip-list and storing it in a new node of level 3, what changes would need to be made to the `links` arrays in the other nodes in the skip-list?



9. Display the result of calling `toString()` for a skip-list of type `SkipList<String, Integer>` that contains each word (the keys) in the sentence below. The values should be the position (an integer) of each word in the sentence. For example, the entry for time would be `("time", 5)` because time is the fifth word in the sentence. What are the values of `size`, `maxCap`, and `maxLevel` for this skip-list?

   Now is a good time for you to decide whether you should major or minor in Computer Science

Programming Projects

1. Complete all methods of class `HuffmanTree` and test them out using a document file and a Java source file on your computer. You can download classes `BitString` and `BitStringBuilder` from the Web site for this textbook.
2. Use a `HashMap` to store the frequency counts for all the words in a large text document. When you are done, display the contents of this `HashMap`. Next, create a set view of the `Map` and store its contents in an array. Then sort the array based on key value and display it. Finally, sort the array in decreasing order by frequency and display it.
3. Solve Project 2 using a `TreeMap`. You can display the words in key sequence without performing a sort.
4. Modify Project 2 to save the line numbers for every occurrence of a word as well as the count.
5. Based on an example in Brian W. Kernighan and Rob Pike, *The Practice of Programming*, Addison-Wesley, 1999, we want to generate "random text" in the style of another author. Your first task is to collect a group of prefix strings of two words that occur in a text file and associate them with a list of suffix strings using a `Map`. For example, the text for Charles Dickens' *A Christmas Carol* contains the four phrases:

   Marley was dead: to begin with.

   Marley was as dead as a doornail.

   Marley was as dead as a doornail.

   Marley was dead.

   The prefix string `"Marley was"` would be associated with the `ArrayList` containing the four suffix strings `"dead:", "as", "as", "dead."`. You must go through the text and examine each successive pair of two-word strings to see whether that pair is already in the map as a key. If so, add the next word to the `ArrayList` that is the value for that prefix string. For example, in examining the first two sentences shown, you would first add to the entry (`"Marley was"`, `ArrayList "dead:"`). Next you would add the entry (`"was dead"`, `ArrayList "as"`). Next you would add the entry (`"dead as"`, `ArrayList "a"`), and so on. When you retrieve the prefix `"Marley was"` again, you would modify the `ArrayList` that is its value, and the entry would become (`"Marley was"`, `ArrayList "dead : ", "as"`). When you are all finished, add the entry `"THE_END"` to the suffix list for the last prefix placed in the `Map`.

   Once you have scanned the complete text, it is time to use the `Map` to begin generating new text that is in the same style as the old text. Output the first prefix you placed in the `Map: "Marley was"`. Then retrieve the `ArrayList` that is the value for this prefix. Randomly select one of the suffixes and then output the suffix. For example, the output text so far might be `"Marley was dead"` if the suffix `"dead"` was selected from the `ArrayList` of suffixes for `"Marley was"`. Now continue with the two-word sequence consisting of the second word from the previous prefix and the suffix (that would be the string `"was dead"`). Look it up in the `Map`, randomly select one of the suffixes and output it. Continue this process until the suffix `"THE_END"` is selected.
6. Complete class `HashtableOpen` so that it fully implements the `Map` interface described in Section 7.2. As part of this, write method `entrySet` and classes `EntrySet` and `SetIterator` as described in Section 7.5. Class `SetIterator` provides methods `hasNext` and `next`. Use data field `index` to keep track of the next value of the iterator (initially 0). Data field `lastItemReturned` keeps track of the index of the last item returned by `next`; this is used by the `remove method`. The `remove` method removes the last item returned by the `next` method from the `Set`. It may only be called once for each call to `next`. Thus, the `remove` method checks to see that `lastItemReturned` has a valid value (not –1) and then sets it to an invalid value (–1) just before returning to the caller.
7. Complete class `HashtableChain` so that it fully implements the `Map` interface, and test it out. Complete class `SetIterator` as described in Project 6.
8. Complete the implementation of class `HashSetOpen`, writing it as an adapter class of `HashtableOpen`.
9. Complete the implementation of class `HashSetChain`, writing it as an adapter class of `HashtableChain`.

10. Revise method `put` for `HashtableOpen` to place a new item into an already deleted spot in the search chain. Don't forget to check the scenario where the key has already been inserted.
11. Update the `SkipList` class to implement the `Map` interface. Change the heading to read

    ```
    public class SkipList<K, V> implements Map<K, V>
    ```

    You should then provide a constructor that takes a `Comparator` parameter, and a no-argument constructor that initializes the `Comparator` to use the natural ordering of the keys. See the implementation of `KWPriorityQueue` in Section 6.6.

Answers to Quick-Check Exercises

1. `Set s = new HashSet<>(Set.of('a', 'b', 'c')); s.add('d');`
2.
```
s.add('a');         // add 'a', duplicate - returns false
s.add('A');         // add 'A', returns true
next = 'b';
s.contains(next);   // 'b' is in the set, returns true
```
3. The value associated with key `1234` is changed to `"Jane Smith"`. The string `"Jane Doe"` is returned.
4. The string `"Jane Doe"` and then **`null`**.
5. `1234` at `234`, `1999` at `999`, `1250` at `250`, `2000` at `000`, `3999` at `001`.
6. `1234` at `234`, `1999` at `999`, `1250` at `250`, `2000` at `000`, `3999` at `003`.
7. `2000` in a linked list at `000`, `1234` in a linked list `at 234`, `1250` in a linked list at `250`, `1999` and `3999` in a linked list at `999`.
8. `AbstractMap`, `AbstractSet`
9. `HashMap` and `TreeMap`, `HashSet` and `TreeSet`
10. `entrySet`, `iterator`
11. `pred[0]` references 15, `pred[1]` references 10 and `pred[2]` references `head`.

Chapter 8

Sorting

Chapter Objectives

- To learn how to use the standard sorting methods in the Java API
- To learn how to implement the following sorting algorithms: selection sort, insertion sort, Shell sort, merge sort, Timsort, heapsort, and quicksort
- To understand the difference in performance of these algorithms, and which to use for small arrays, which to use for medium arrays, and which to use for large arrays

Sorting is the process of rearranging the data in an array or a list so that it is in increasing (or decreasing) order. Because sorting is done so frequently, computer scientists have devoted much time and effort to developing efficient algorithms for sorting arrays. Even though many languages (including Java) provide sorting utilities, it is still very important to study these algorithms because they illustrate several well-known ways to solve the sorting problem, each with its own merits. You should know how they are written so that you can duplicate them if you need to use them with languages that don't have sorting utilities.

Another reason for studying these algorithms is that they illustrate some very creative approaches to problem-solving. For example, the insertion sort algorithm adapts an approach used by card players to arrange a hand of cards; the merge sort algorithm builds on a technique used to sort external data files. Several algorithms use divide-and-conquer to break a larger problem into more manageable subproblems. The Shell sort is a very efficient sort that works by sorting many small subarrays using insertion sort, which is a relatively inefficient sort when used by itself. The merge sort and quicksort algorithms are both recursive. Method `heapsort` uses a heap as its underlying data structure. The final reason for studying sorting is to learn how computer scientists analyze and compare the performance of several different algorithms that perform the same operation.

We will cover two quadratic ($O(n^2)$) sorting algorithms that are fairly simple and appropriate for sorting small arrays but are not recommended for large arrays. We will also discuss four sorting algorithms that give improved performance ($O(n \log n)$) on large arrays and one that gives performance that is much better than $O(n^2)$ but not as good as $O(n \log n)$.

Our goal is to provide a sufficient selection of quadratic sorts and faster sorts. A few other sorting algorithms are described in the programming projects. Our expectation is that your instructor will select which algorithms you should study.

Sorting

8.1 Using Java Sorting Methods

The Java API `java.util` provides a class `Arrays` with several overloaded sort methods for different array types. In addition, the class `Collections` (also part of the API `java.util`) contains similar sorting methods for `Lists`. The methods for arrays of primitive types are based on the quicksort algorithm (Section 8.9), and the methods for arrays of `Objects` and for `Lists` are based on the Timsort algorithm (Section 8.7). Both algorithms are O(n log n).

Method `Arrays.sort` is defined as a **`public static void`** method and is overloaded (see Table 8.1). The first argument in a call can be an array of any primitive type (although we have just shown **`int`**`[]`) or an array of objects. The first two method signatures shown in the table are for an array of primitive types.

For an array of objects, the class type of the object must implement the `Comparable` interface. This means that it must provide a `compareTo` method that specifies the natural ordering of two objects being compared. The third and fourth method signatures are used for sorting arrays of objects using their natural ordering.

Optionally, a `Comparator` object (see Section 6.6) may be passed as the last argument as shown in the last two method signatures. If a `Comparator` is passed, its `compare` method will be used to determine the ordering instead of method `compareTo`.

In class `Arrays`, the two methods that use a `Comparator` are *generic methods*. We showed how to declare generic methods in Section 5.3. Generic methods, such as generic classes, have parameters. The generic parameter(s) precede the method type. For example, in the declaration

```
public static <T> void sort(T[] items, Comparator<? super T> comp)
```

`T` represents the generic parameter for the `sort` method and should appear in the method's parameter list (e.g., `T[] items`). For the second method parameter, the notation

TABLE 8.1
Methods `sort` in Classes `java.util.Arrays`

Method sort in Class Arrays	Behavior
`public static void sort(int[] items)`	Sorts the array `items` in ascending order
`public static void sort(int[] items, int fromIndex, int toIndex)`	Sorts array elements `items[fromIndex]` to `items [toIndex]` in ascending order
`public static void sort(Object[] items)`	Sorts the objects in array `items` in ascending order using their natural ordering (defined by method `compareTo`). All objects in `items` must implement the `Comparable` interface and must be mutually comparable
`public static void sort(Object[] items, int fromIndex, int toIndex)`	Sorts array elements `items[fromIndex]` to `items[toIndex]` in ascending order using their natural ordering (defined by method `compareTo`). All objects must implement the `Comparable` interface and must be mutually comparable
`public static <T> void sort(T[] items, Comparator<? super T> comp)`	Sorts the objects in `items` in ascending order as defined by method `comp.compare`. All objects in `items` must be mutually comparable using method `comp.compare`
`public static <T> void sort(T[] items, int fromIndex, int toIndex, Comparator<? super T> comp)`	Sorts the objects in `items[fromIndex]` to `items[toIndex]` in ascending order as defined by method `comp.compare`. All objects in `items` must be mutually comparable using method `comp.compare`

`Comparator<? super T> comp` means that `comp` must be an object that implements the `Comparator` interface for type `T` or for a superclass of type `T`. For example, you can define a class that implements `Comparator<Number>` and use it to sort an array of `Integer` objects or an array of `Double` objects.

EXAMPLE 8.1 If array `items` stores a collection of integers, the method call

```
Arrays.sort(items);
```

sorts all the elements in array `items`. The method call

```
Arrays.sort(items, 0, items.length / 2) ;
```

sorts the integers in the first half of the array, leaving the second half of the array unchanged.

EXAMPLE 8.2 Let's assume class `Person` is defined as follows:

```
public class Person implements Comparable<Person> {
    // Data Fields
    /* The last name */
    private String lastName;
    /* The first name */
    private String firstName;
    /* Birthday represented by an integer from 1 to 366 */
    private int birthDay;

    // Methods
    /** Compares two Person objects based on names. The result
            is based on the last names if they are different;
```

```
             otherwise, it is based on the first names.
        @param obj The other Person
        @return A negative integer if this person's name
                precedes the other person's name;
                0 if the names are the same;
                a positive integer if this person's name follows
                the other person's name.
     */
    @Override
    public int compareTo(Person other) {
        // Compare this Person to other using last names.
        int result = lastName.compareTo(other.lastName);
        // Compare first names if last names are the same.
        if (result == 0)
            return firstName.compareTo(other.firstName);
        else
            return result;
    }

    // Other methods
    ...
}
```

Method `Person.compareTo` compares two `Person` objects based on their names using the last name as the primary key and the first name as the secondary key (the natural ordering). If `people` is an array of `Person` objects, the statement

```
Arrays.sort(people);
```

sorts the array `people` based on the names of the people in the array. Although the sort operation is $O(n \log n)$, the comparison of two names is $O(k)$, where k is the length of the shorter name.

EXAMPLE 8.3 You can also use a class that implements `Comparator<Person>` to compare `Person` objects. As an example, method `compare` in class `ComparePerson` compares two `Person` objects based on their birthdays, not their names.

```
import java.util.Comparator;

public class ComparePerson implements Comparator<Person> {
    /** Compare two Person objects based on birth date.
        @param left The left-hand side of the comparison
        @param right The right-hand side of the comparison
        @return A negative integer if the left person's birthday
                precedes the right person's birthday;
                0 if the birthdays are the same;
                a positive integer if the left person's birthday
                follows the right person's birthday.
     */
    @Override
    public int compare(Person left, Person right) {
        return left.getBirthDay() - right.getBirthDay();
    }
}
```

Now we can use the statement

```
Arrays.sort(people, new ComparePerson());
```

to sort the array `people` based on the birthdays of the people in the array. Comparing two birthdays is an O(1) operation.

Passing a Lambda Expression as a Comparator

Rather than writing class ComparePerson, we could pass a lambda expression as a Comparator object to the sort method. We can sort using the following statement, where the lambda expression specifies the compare method to be used with array people.

```
Arrays.sort(people, (p1, p2) -> p1.getBirthday() - p2.getBirthday());
```

Collections.sort Methods

We can use the sorting methods in Collections.sort (see Table 8.2) to sort a collection of objects that implement the List interface (e.g., an ArrayList or a LinkedList). If only one argument is provided, the objects in the list must implement the Comparable interface. In this case, method compareTo is called by the sorting method to determine the relative ordering of two objects. Rather than rearranging the elements in the list, method Collections.sort first copies the list elements to an array, sorts the array using Arrays.sort, and then copies them back to the list.

Optionally, a Comparator can be passed as a second argument. Using a Comparator, you can compare objects based on some other information rather than using their natural ordering.

If peopleList is a List of Person objects, the statement

```
Collections.sort(peopleList);
```

sorts the list peopleList based on the names of the people in the list (natural ordering).

The statement

```
Collections.sort(peopleList, new ComparePerson());
```

sorts the list peopleList based on their birthdays.

Method List.sort

We described the sort methods in the Collections class for sorting lists because many Java programmers still use them. A new List.sort method became available in Java 8 (see Table 8.3). The List.sort is not a static method and can be applied to an object that satisfies

TABLE 8.2
Methods sort in Classes Collections.util

Method sort in Class Collections	Behavior
`public static <T extends Comparable<T>> void sort(List<T> list)`	Sorts the objects in list in ascending order using their natural ordering (defined by method compareTo). All objects in list must implement the Comparable interface and must be mutually comparable
`public static <T> void sort (List<T> list, Comparator<? super T> comp)`	Sorts the objects in list in ascending order as defined by method comp.compare. All objects must be mutually comparable

TABLE 8.3
Method sort in Interface List

Method sort in Interface List	Behavior
`default void sort(Comparator<? super E> comp)`	Sorts the objects in the list in ascending order as defined by method comp.compare. All objects must be mutually comparable. If comp is **null**, the natural ordering is used

the `List` interface. It takes a single argument, which is a `Comparator` object. For example, the statement

```
peopleList.sort(new ComparePerson());
```

sorts `peopleList` based on the age of the people in the list. If we want to arrange the objects in `peopleList` based on their natural ordering, we must use the following:

```
peopleList.sort(null);
```

EXERCISES FOR SECTION 8.1

SELF-CHECK

1. Indicate whether each of the following method calls is valid. Describe why it isn't valid or, if it is valid, describe what it does. Assume `people` is an array of `Person` objects and `peopleList` is a `List` of `Person` objects.
   a. `people.sort();`
   b. `Arrays.sort(people, 0, people.length - 3);`
   c. `Arrays.sort(peopleList, 0, peopleList.length - 3);`
   d. `Collections.sort(people);`
   e. `Collections.sort(peopleList, new ComparePerson());`
   f. `Collections.sort(peopleList, 0, peopleList.size() - 3);`

PROGRAMMING

1. Write a method `call` to sort the last half of array `people` using the natural ordering.
2. Write a method `call` to sort the last half of array `people` using the ordering determined by class `ComparePerson`.
3. Write a method `call` to sort `peopleList` using the natural ordering.

8.2 Selection Sort

Selection sort is a relatively easy-to-understand algorithm that sorts an array by making several passes through the array, *selecting* the next smallest item in the array each time, and placing it where it belongs in the array. We illustrate all sorting algorithms using an array of integer values for simplicity. However, each algorithm sorts an array of `Comparable` objects, so the **`int`** values must be wrapped in `Integer` objects.

We show the algorithm next, where n is the number of elements in an array with subscripts 0 through $n - 1$ and `fill` is the subscript of the element that will store the next smallest item in the array.

Selection Sort Algorithm

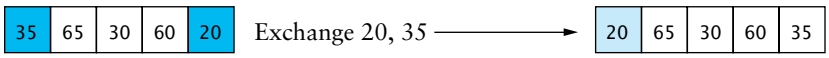
1. **`for`** `fill` = 0 to n – 2 do
2. Set `posMin` to the subscript of the smallest item in the subarray starting at subscript `fill`.
3. Exchange the item at `posMin` with the one at `fill`.

Step 2 involves a search for the smallest item in each subarray. It requires a loop in which we compare each element in the subarray, starting with the one at position `fill + 1`, with the smallest value found so far. In the refinement of Step 2 shown in the following algorithm (Steps 2.1 through 2.4), we use `posMin` to store the subscript of the smallest value found so far. We assume that its initial position is `fill`.

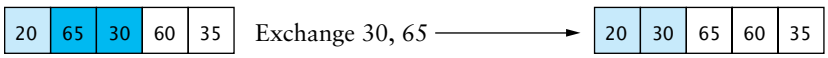
Refinement of Selection Sort Algorithm (Step 2)

2.1 Initialize `posMin` to `fill`.
2.2 `for next = fill + 1 to n - 1`
2.3 `if` the item at `next` is less than the item at `posMin`
2.4 Reset `posMin` to next.

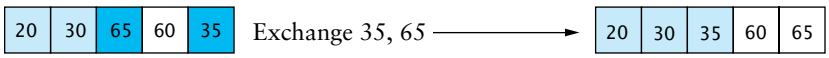
First the selection sort algorithm finds the smallest item in the array (smallest is 20) and moves it to position 0 by exchanging it with the element currently at position 0. At this point, the sorted part of the array consists of the new element at position 0. The values to be exchanged are dark in all diagrams. The sorted elements are lightly shaded.



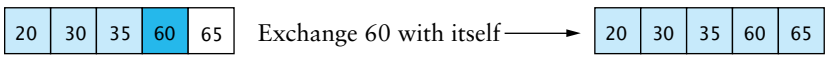
Next, the algorithm finds the smallest item in the subarray starting at position 1 (next smallest is 30) and exchanges it with the element currently at position 1:



At this point, the sorted portion of the array consists of the elements at positions 0 and 1. Next, the algorithm selects the smallest item in the subarray starting at position 2 (next smallest is 35) and exchanges it with the element currently at position 2:



At this point, the sorted portion of the array consists of the elements at positions 0, 1, and 2. Next, the algorithm selects the smallest item in the subarray starting at position 3 (next smallest is 60) and exchanges it with the element currently at position 3:



The element at position 4, the last position in the array, must store the largest value (largest is 65), so the array is sorted.

Analysis of Selection Sort

Steps 2 and 3 are performed $n - 1$ times. Step 3 performs an exchange of items; consequently, there are $n - 1$ exchanges.

Step 2.3 involves a comparison of items and is performed ($n - 1 -$ *fill*) times for each value of *fill*. Since *fill* takes on all values between 0 and $n - 2$, the following series computes the number of executions of Step 2.3:

$$(n-1)+(n-2)+\cdots+3+2+1$$

This is a well-known series that can be written in closed form as

$$\frac{n\times(n-1)}{2}=\frac{n^2}{2}-\frac{n}{2}$$

For very large n, we can ignore all but the most significant term in this expression, so the number of comparisons is $O(n^2)$ and the number of exchanges is $O(n)$. Because the number of comparisons increases with the square of n, the selection sort is called a *quadratic sort*.

Implementation of Selection Sort

Listing 8.1 shows the code for selection sort. In the comments, `table[0 . . fill - 1]` denotes array elements `table[0]` through `table[fill - 1]`.

LISTING 8.1
SelectionSort.java

```
/** Implements the selection sort algorithm. */
public class SelectionSort {

    /** Sort the array using selection sort algorithm.
        pre: table contains Comparable objects.
        post: table is sorted.
        @param table The array to be sorted
     */
    public static void sort(Object[] table) {
        int n = table.length;
        for (int fill = 0; fill < n - 1; fill++) {
            // Invariant: table[0 . . fill - 1] is sorted.
            int posMin = fill;
            for (int next = fill + 1; next < n; next++) {
                // Invariant: table[posMin] is the smallest item in
                // table[fill . . next - 1].
                if (((Comparable) table[next]).compareTo(table[posMin]) < 0) {
                      posMin = next;
                }
            }
            // assert: table[posMin] is the smallest item in
            // table[fill . . n - 1].
            // Exchange table[fill] and table[posMin].
            var temp = table[fill];
            table[fill] = table[posMin];
            table[posMin] = temp;
            // assert: table[fill] is the smallest item in
            // table[fill . . n - 1].
        }
        // assert: table[0 . . n - 1] is sorted.
    }
}
```

PROGRAM STYLE

Making Sort Methods Generic

The selection sort method shown in Listing 8.1 will sort any array of `Comparable` objects using method `compareTo`. Its heading matches the signature in Table 8.1 for sorting arrays using their natural ordering. We could simplify the method by substituting `Comparable` for `Object` in the method heading:

```
public static void (Comparable[] table) {
```

This would eliminate the need to cast the array elements to `Comparable` before comparing them.

Either version of the selection sort will compile, but this one will generate a warning message regarding an unchecked or unsafe call to `compareTo`. You can eliminate this warning message by making the sort a generic sort:

```
public static <T extends Comparable<T>> void sort (T[] table) {
```

where the generic type parameter, `T`, must implement the `Comparable<T>` interface. This is the approach we will follow in the rest of the chapter.

Finally, if we needed to write a generic selection sort method that uses a `Comparator` object, we would write the method heading as

```
public static <T> void sort(T[] table, Comparator<? super T> comp) {
```

where `<? Super T>` means that the `Comparator` object `comp` is of type `T` or of a type that is a superclass of `T`. Either `T` or its superclass must define a `compare` method.

EXERCISES FOR SECTION 8.2

SELF-CHECK

1. Show the progress of each pass of the selection sort for the following array. How many passes are needed? How many comparisons are performed? How many exchanges? Show the array after each pass.

   40 35 80 75 60 90 70 75 50 22

2. How would you modify selection sort to arrange an array of values in decreasing sequence?

3. It is not necessary to perform an exchange if the next smallest element is already at position `fill`. Modify the selection sort algorithm to eliminate the exchange of an element with itself. How does this affect big-O for exchanges? Discuss whether the time saved by eliminating unnecessary exchanges would exceed the cost of these extra steps.

PROGRAMMING

1. Modify the selection sort method to sort the elements in decreasing order and to incorporate the change in Self-Check Exercise 3.

2. Add statements to trace the progress of selection sort. Display the array contents after each exchange.

8.3 Insertion Sort

Our next quadratic sorting algorithm, *insertion sort*, is based on the technique used by card players to arrange a hand of cards. The player keeps the cards that have been picked up so far in sorted order. When the player picks up a new card, the player makes room for the new card and then *inserts* it in its proper place.

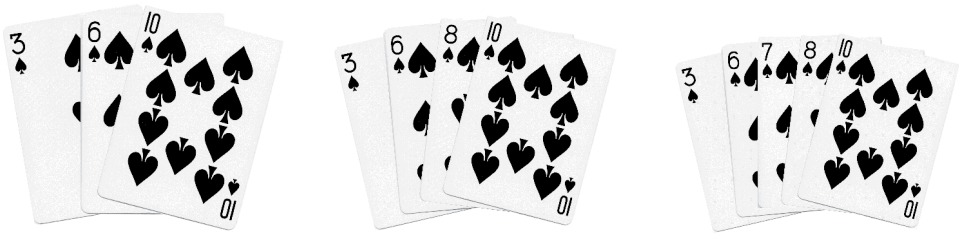


FIGURE 8.1
Picking Up a Hand of Cards

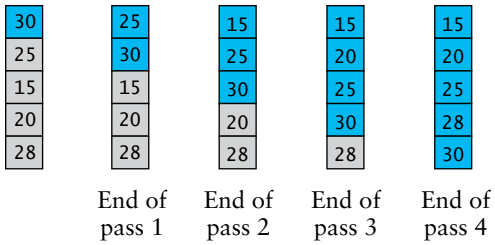


FIGURE 8.2
An Insertion Sort

The left diagram of Figure 8.1 shows a hand of cards (ignoring suits) after three cards have been picked up. If the next card is an 8, it should be inserted between the 6 and 10, maintaining the numerical order (middle diagram). If the next card is a 7, it should be inserted between the 6 and 8 as shown on the right in Figure 8.1.

To adapt this insertion algorithm to an array that has been filled with data, we start with a sorted subarray consisting of the first element only. For example, in the leftmost array of Figure 8.2, the initial sorted subarray consists of only the first value 30 (in element 0). The array element(s) that are in order after each pass are in color, and the elements waiting to be inserted are in gray. We first *insert* the second element (25). Because it is smaller than the element in the sorted subarray, we insert it before the old first element (30), and the sorted subarray has two elements (25, 30 in second diagram). Next, we *insert* the third element (15). It is also smaller than all the elements in the sorted subarray, so we insert it before the old first element (25), and the sorted subarray has three elements (15, 25, 30 in third diagram). Next, we *insert* the fourth element (20). It is smaller than the second and third elements in the sorted subarray, so we insert it before the old second element (25), and the sorted subarray has four elements (15, 20, 25, 30 in the fourth diagram). Finally, we insert the last element (28). It is smaller than the last element in the sorted subarray, so we insert it before the old last element (30), and the array is sorted. The algorithm follows.

Insertion Sort Algorithm

1. **for** each array element from the second (`nextPos = 1`) to the last
2. Insert the element at nextPos where it belongs in the array, increasing the length of the sorted subarray by 1 element.

To accomplish Step 2, the insertion step, we need to make room for the element to be inserted (saved in `nextVal`) by shifting all values that are larger than it, starting with the last value in the sorted subarray.

Refinement of Insertion Sort Algorithm (Step 2)

2.1 `nextPos` is the position of the element to insert.

2.2 Save the value of the element to insert in `nextVal`.

2.3 `while nextPos > 0` and the element at `nextPos - 1 > nextVal`
2.4 Shift the element at `nextPos - 1` to position `nextPos`.
2.5 Decrement nextPos by 1.
2.6 Insert `nextVal` at `nextPos`.

We illustrate these steps in Figure 8.3. For the array shown on the left, the first three elements (positions 0, 1, and 2) are in the sorted subarray, and the next element to insert is 20. First we save 20 in `nextVal` and 3 in `nextPos`. Next we shift the value in position 2 (30) down one position (see the second array in Figure 8.3), and then we shift the value in position 1 (25) down one position (see third array in Figure 8.3). After these shifts (third array), there will temporarily be two copies of the last value shifted (25). The first of these (white background in Figure 8.3) is overwritten when the value in `nextVal` is moved into its correct position (`nextPos` is 1). The four-element sorted subarray is shaded in color on the right of Figure 8.3.

Analysis of Insertion Sort

The insertion step is performed $n - 1$ times. In the worst case, all elements in the sorted subarray are compared to `nextVal` for each insertion, so the maximum number of comparisons is represented by the series

$$1 + 2 + 3 + \cdots + (n-2) + (n-1)$$

which is $O(n^2)$. In the best case (when the array is already sorted), only one comparison is required for each insertion, so the number of comparisons is $O(n)$. The number of shifts performed during an insertion is one less than the number of comparisons or, when the new value is the smallest so far, the same as the number of comparisons. However, a shift in an insertion sort requires the movement of only one item, whereas in a selection sort, an exchange involves a temporary item and requires the movement of three items. A Java array of objects contains references to the actual objects, and it is these references that are changed. The actual objects remain in the physical locations where they were first created.

Implementation of Insertion Sort

Listing 8.2 shows the `InsertionSort`. We use method `insert` to perform the insertion step shown earlier. It would be more efficient to insert this code inside the **`for`** statement; however, using a method will make it easier to implement the Shell sort algorithm later.

The **`while`** statement in method `insert` compares and shifts all values greater than `nextVal` in the subarray `table[0 .. nextPos - 1]`. The **`while`** condition

```
((nextPos > 0) && (nextVal.compareTo(table[nextPos - 1]) < 0))
```

causes loop exit if the first element has been moved or if `nextVal` is not less than the next element to move. The order of the `&&` operand is critical. Recall that Java performs short-circuit evaluation. If the left-hand operand of an `&&` operation is false, the right-hand operand is

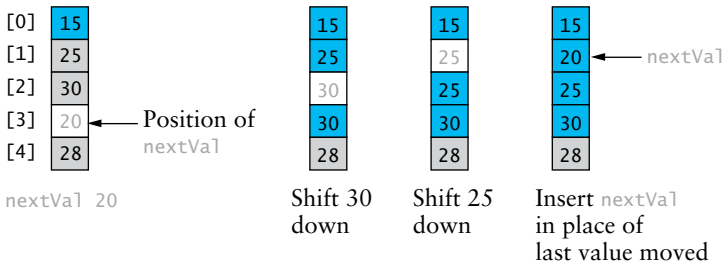


FIGURE 8.3
Inserting the Fourth Array Element

not evaluated. If this were not the case, when `nextPos` becomes 0, the array subscript would be –1, which throws an `ArrayIndexOutOfBoundsException`.

LISTING 8.2

InsertionSort.java

```
/** Implements the insertion sort algorithm. */
public class InsertionSort {

    /** Sort the table using insertion sort algorithm.
        pre: table contains Comparable objects.
        post: table is sorted.
        @param table The array to be sorted
     */
    public static <T extends Comparable<T>> void sort(T[] table) {
        for (int nextPos = 1; nextPos < table.length; nextPos++) {
            // Invariant: table[0 . . nextPos - 1] is sorted.
            // Insert element at position nextPos
            // in the sorted subarray.
            insert(table, nextPos);
        } // End for.
    } // End sort.

    /** Insert the element at nextPos where it belongs in the array.
        pre: table[0 . . nextPos - 1] is sorted.
        post: table[0 . . nextPos] is sorted.
        @param table The array being sorted
        @param nextPos The position of the element to insert
     */
    private static <T extends Comparable<T>> void insert(T[] table, int nextPos) {
        T nextVal = table[nextPos];
        // Element to insert.
        while (nextPos > 0 && nextVal.compareTo(table [nextPos - 1]) < 0) {
            table[nextPos] = table[nextPos - 1];
            // Shift down.
            nextPos-- ;
            //  Check next smaller element.
        }
        // Insert nextVal at nextPos.
        table[nextPos] = nextVal;
    }
}
```

EXERCISES FOR SECTION 8.3

SELF-CHECK

1. Sort the following array using insertion sort. How many passes are needed? How many comparisons are performed? How many exchanges? Show the array after each pass.

   40 35 80 75 60 90 70 75 50 22

PROGRAMMING

1. Eliminate method `insert` in Listing 8.2 and write its code inside the **`for`** statement.
2. Add statements to trace the progress of insertion sort. Display the array contents after the insertion of each value.

8.4 Comparison of Quadratic Sorts

Table 8.4 summarizes the performance of two quadratic sorts. To give you some idea as to what these numbers mean, Table 8.5 shows some values of n and n^2. If n is small (say, 100 or less), it really doesn't matter which sorting algorithm you use. Insertion sort gives the best performance for larger arrays. Insertion sort is better because it takes advantage of any partial sorting that is in the array and uses less costly shifts instead of exchanges to rearrange array elements. In the next section, we discuss a variation on insertion sort, known as Shell sort, that has $O(n^{3/2})$ or better performance.

Since the time to sort an array of n elements is proportional to n^2, none of these algorithms is particularly good for large arrays (i.e., $n > 100$). The best sorting algorithms provide n log n average-case behavior and are considerably faster for large arrays. In fact, one of the algorithms that we will discuss has n log n worst-case behavior. You can get a feel for the difference in behavior by comparing the last column of Table 8.5 with the middle column.

Recall from Section 2.1 that big-O analysis ignores any constants that might be involved or any overhead that might occur from method calls needed to perform an exchange or a comparison. However, the tables give you an estimate of the relative performance of the different sorting algorithms.

We haven't talked about storage usage for these algorithms. Both quadratic sorts require storage for the array being sorted. However, there is only one copy of this array, so the array is sorted in place. There are also requirements for variables that store references to particular elements, loop control variables, and temporary variables. However, for large n, the size of the array dominates these other storage considerations, so the extra space usage is proportional to $O(1)$.

TABLE 8.4
Comparison of Quadratic Sorts

	Number of Comparisons		Number of Exchanges	
	Best	Worst	Best	Worst
Selection sort	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n)$
Insertion sort	$O(n)$	$O(n^2)$	$O(1)$	$O(n^2)$

TABLE 8.5
Comparison of Rates of Growth

n	n^2	n log n
8	64	24
16	256	64
32	1024	160
64	4096	384
128	16,384	896
256	65,536	2048
512	262,144	4608

Comparisons versus Exchanges

We have analyzed comparisons and exchanges separately, but you may be wondering whether one is more costly (in terms of computer time) than the other. In Java, an exchange requires your computer to switch two object references using a third object reference as an intermediary. A comparison requires your computer to execute a `compareTo` method. The cost of a comparison depends on its complexity, but it will probably be more costly than an exchange because of the overhead to call and execute method `compareTo`. In some programming languages (but not Java), an exchange may require physically moving the information in each object rather than simply swapping object references. For these languages, the cost of an exchange would be proportional to the size of the objects being exchanged and may be more costly than a comparison.

EXERCISES FOR SECTION 8.4

SELF-CHECK

1. Complete Table 8.5 for n = 1024 and n = 2048.
2. What do the new rows of Table 8.5 tell us about the increase in time required to process an array of 1024 elements versus an array of 2048 elements for O(n), O(n^2), and O(n log n) algorithms?

8.5 Shell Sort: A Better Insertion Sort

Next, we describe the *Shell sort,* which is a type of insertion sort but with O($n^{3/2}$) or better performance. Shell sort is named after its discoverer, Donald L. Shell ("A High-Speed Sorting Procedure," *Communications of the ACM,* Vol. 2, No. 7 [1959], pp. 30–32). You can think of the Shell sort as a divide-and-conquer approach to insertion sort. Instead of sorting the entire array at the start, the idea behind Shell sort is to sort many smaller subarrays using insertion sort before sorting the entire array. The initial subarrays will contain two or three elements, so the insertion sorts will go very quickly. After each collection of subarrays is sorted, a new collection of subarrays with approximately twice as many elements as before will be sorted. The last step is to perform an insertion sort on the entire array, which has been presorted by the earlier sorts.

As an example, let's sort the following array using initial subarrays with only two and three elements. We determine the elements in each subarray by setting a `gap` value between the subscripts in each subarray. We will explain how we pick the `gap` values later. We will use an initial `gap` of 7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	35	80	75	60	90	70	75	55	90	85	34	45	62	57	65

A `gap` of 7 means the first subarray has subscripts 0, 7, 14 (element values 40, 75, 57, light color); the second subarray has subscripts 1, 8, 15 (element values 35, 55, 65, darker color); the third subarray has subscripts 2, 9 (element values 80, 90, gray); and so on. There are seven subarrays. We start the process by inserting the value at position 7 (value of `gap`) into its subarray (elements at 0 and 7). Next, we insert the element at position 8 into its subarray

(elements at 1 and 8). We continue until we have inserted the last element (at position 15) in its subarray (elements at 1, 8, and 15). The result of performing insertion sort on all seven subarrays with two or three elements follows:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	35	80	75	34	45	62	57	55	90	85	60	90	70	75	65

Next, we use a gap of 3. There are only three subarrays, and the longest one has six elements. The first subarray has subscripts 0, 3, 6, 9, 12, 15; the second subarray has subscripts 1, 4, 7, 10, 13; the third subarray has subscripts 2, 5, 8, 11, 14.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	35	80	75	34	45	62	57	55	90	85	60	90	70	75	65

We start the process by inserting the element at position 3 (value of gap) into its subarray. Next, we insert the element at position 4, and so on. The result of all insertions follows:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
40	34	45	62	35	55	65	57	60	75	70	75	90	85	80	90

Finally, we use a gap of 1, which performs an insertion sort on the entire array. Because of the presorting, it will require 1 comparison to insert 34, 1 comparison to insert 45 and 62, 4 comparisons to insert 35, 2 comparisons to insert 55, 1 comparison to insert 65, 3 comparisons to insert 57, 1 comparison to insert 60, and only 1 or 2 comparisons to insert each of the remaining values except for 80 (3 comparisons).

The algorithm for Shell sort follows. Steps 2 through 4 correspond to the insertion sort algorithm shown earlier. Because the elements with subscripts 0 through `gap - 1` are the first elements in their subarrays, we begin Step 4 by inserting the element at position `gap` instead of at position 1 as we did for the insertion sort. Step 1 sets the initial `gap` between subscripts to `n/2`, where `n` is the number of array elements. To get the next `gap` value, Step 6 divides the current `gap` value by 2.2 (chosen by experimentation). We want the `gap` to be 1 during the last insertion sort so that the entire array will be sorted. Step 5 ensures this by resetting the `gap` to 1 if it is 2.

Shell Sort Algorithm

1. Set the initial value of `gap` to n/2.
2. **`while`** `gap` > 0
3. **`for`** each array element from position `gap` to the last element
4. Insert this element where it belongs in its subarray.
5. **`if`** `gap` is 2, set it to 1.
6. **`else`** `gap = gap/2.2`.

Refinement of Step 4, the Insertion Step

4.1 `nextPos` is the position of the element to insert.
4.2 Save the value of the element to insert in `nextVal`.
4.3 **`while`** `nextPos > gap` and the element at `nextPos - gap > nextVal`
4.4 Shift the element at `nextPos - gap` to position `nextPos`.
4.5 Decrement `nextPos` by `gap`.
4.6 Insert `nextVal` at `nextPos`.

Analysis of Shell Sort

You may wonder why Shell sort is an improvement over regular insertion sort, because it ends with an insertion sort of the entire array. Each later sort (including the last one) will be performed on an array whose elements have been presorted by the earlier sorts. Because the behavior of insertion sort is closer to $O(n)$ than $O(n^2)$ when an array is nearly sorted, the presorting will make the later sorts, which involve larger subarrays, go more quickly. As a result of presorting, only 19 comparisons were required to perform an insertion sort on the last 15-element array shown in the previous section. This is critical because it is precisely for larger arrays where $O(n^2)$ behavior would have the most negative impact. For the same reason, the improvement of Shell sort over insertion sort is much more significant for large arrays.

A general analysis of Shell sort is an open research problem in computer science. The performance depends on how the decreasing sequence of values for `gap` is chosen. It is known that Shell sort is $O(n^2)$ if successive powers of 2 are used for `gap` (i.e., 32, 16, 8, 4, 2, 1). If successive values for `gap` are of the form $2^k - 1$ (i.e., 31, 15, 7, 3, 1), however, it can be proven that the performance is $O(n^{3/2})$. This sequence is known as *Hibbard's sequence*. There are other sequences that give similar or better performance.

We have presented an algorithm that selects the initial value of `gap` as $n/2$ and then divides by 2.2 and truncates to the next lowest integer. Empirical studies of this approach show that the performance is $O(n^{5/4})$ or maybe even $O(n^{7/6})$, but there is no theoretical basis for this result (M. A. Weiss, *Data Structures and Problem Solving Using Java* [Addison-Wesley, 1998], p. 230).

Implementation of Shell Sort

Listing 8.3 shows the code for Shell sort. Method `insert` has a third parameter, `gap`. The expression after `&&`

```
((nextPos > gap - 1) && (nextVal.compareTo(table[nextPos - gap]) < 0))
```

compares elements that are separated by the value of `gap` instead of by 1. The expression before `&&` is false if `nextPos` is the subscript of the first element in a subarray. The statements in the **`while`** loop shift the element at `nextPos` down by `gap` (one position in the subarray) and reset `nextPos` to the subscript of the element just moved.

LISTING 8.3
ShellSort.java

```
/** Implements the Shell sort algorithm. */
public class ShellSort {
    /** Sort the table using Shell sort algorithm.
        pre: table contains Comparable objects.
        post: table is sorted.
        @param table The array to be sorted
     */
    public static <T extends Comparable<T>> void sort(T[] table) {
        // Gap between adjacent elements.
        int gap = table.length / 2;
        while (gap > 0) {
            for (int nextPos = gap; nextPos < table.length; nextPos++) {
                // Insert element at nextPos in its subarray.
                insert(table, nextPos, gap);
            } // End for.
```

```
            // Reset gap for next pass.
            if (gap == 2) {
                gap = 1;
            } else {
                gap = (int) (gap / 2.2);
            }
        } // End while.
    } // End sort.

    /** Inserts element at nextPos where it belongs in array.
        pre: Elements through nextPos - gap in subarray are sorted.
        post: Elements through nextPos in subarray are sorted.
        @param table The array being sorted
        @param nextPos The position of element to insert
        @param gap The gap between elements in the subarray
     */
    private static <T extends Comparable<T>> void insert(T[] table,
                                                         int nextPos, int gap) {
        T nextVal = table[nextPos];
        // Element to insert.
        // Shift all values > nextVal in subarray down by gap.
        while ((nextPos > gap - 1) && (nextVal.compareTo
                                       (table [nextPos - gap]) < 0)) {
            // First element not shifted.
            table[nextPos] = table[nextPos - gap];
            // Shift down.
            nextPos -= gap;
            // Check next position in subarray.
        }
        table[nextPos] = nextVal;
        // Insert nextVal.
    }
}
```

EXERCISES FOR SECTION 8.5

SELF-CHECK

1. Trace the execution of Shell sort on the following array. Show the array after all sorts when the gap is 5, the gap is 2, and after the final sort when the gap is 1. List the number of comparisons and exchanges required when the gap is 5, when the gap is 2, and when the gap is 1. Compare this with the number of comparisons and exchanges that would be required for a regular insertion sort.

   40 35 80 75 60 90 70 65 50 22

2. For the example of Shell sort shown in this section, determine how many comparisons and exchanges are required to insert all the elements for each gap value. Compare this with the number of comparisons and exchanges that would be required for a regular insertion sort.

PROGRAMMING

1. Eliminate method `insert` in Listing 8.3 and write its code inside the **`for`** statement.
2. Add statements to trace the progress of Shell sort. Display each value of gap, and display the array contents after all subarrays for that gap value have been sorted.

8.6 Merge Sort

The next algorithm that we will consider is called *merge sort*. A *merge* is a common data processing operation that is performed on two sequences of data (or data files) with the following characteristics:

- Both sequences contain items with a common `compareTo` method.
- The objects in both sequences are ordered in accordance with this `compareTo` method (i.e., both sequences are sorted).

The result of the merge operation is to create a third sequence that contains all of the objects from the first two sorted sequences. For example, if the first sequence is 3, 5, 8, 15 and the second sequence is **4**, **9**, **12**, **20**, the final sequence will be 3, **4**, 5, 8, **9**, **12**, 15, **20**. The algorithm for merging the two sequences follows.

Merge Algorithm

1. Access the first item from both sequences.
2. **`while`** not finished with either sequence
3. Compare the current items from the two sequences, copy the smaller current item to the output sequence, and access the next item from the input sequence whose item was copied.
4. Copy any remaining items from the first sequence to the output sequence.
5. Copy any remaining items from the second sequence to the output sequence.

The **`while`** loop (Step 2) merges items from both input sequences to the output sequence. The current item from each sequence is the one that has been most recently accessed but not yet copied to the output sequence. Step 3 compares the two current items and copies the smaller one to the output sequence. If input sequence A's current item is the smaller one, the next item is accessed from sequence A and becomes its current item. If input sequence B's current item is the smaller one, the next item is accessed from sequence B and becomes its current item. After the end of either sequence is reached, Step 4 or Step 5 copies the items from the other sequence to the output sequence. Note that either Step 4 or Step 5 is executed, but not both.

As an example, consider the sequences shown in Figure 8.4. Steps 2 and 3 will first copy the items from sequence A with the values **244** and **311** to the output sequence; then items from sequence B with values 324 and 415 will be copied; and then the item from sequence A with value **478** will be copied. At this point, we have copied all items in sequence A, so we exit the **`while`** loop and copy the remaining items from sequence B (499, 505) to the output (Steps 4 and 5).

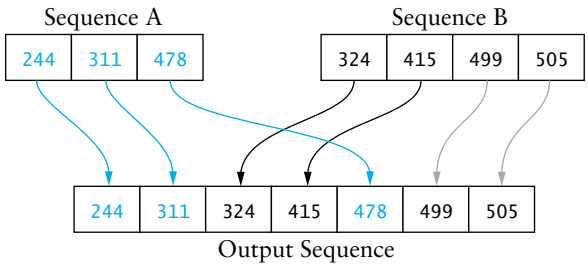


FIGURE 8.4
Merge Operation

Analysis of Merge

For two input sequences that contain a total of n elements, we need to move each element from its input sequence to its output sequence, so the time required for a merge is O(n). How about the space requirements? We need to be able to store both initial sequences and the output sequence. So the array cannot be merged in place, and the *additional* space usage is O(n).

Implementation of Merge

Listing 8.4 shows the merge algorithm applied to arrays of `Comparable` objects. Algorithm Steps 4 and 5 are implemented as `while` loops at the end of the method. The parameter `dest` is the starting position in the output array for the merged elements. Normally, it will be 0, but we provided it as a parameter so that we could reuse the method in cases where it would not be 0.

LISTING 8.4
`Merge` Method

```
/** Merge two sequences.
    pre: leftSequence and rightSequence are sorted.
    post: outputSequence is the merged result and is sorted.
    @param outputSequence The destination
    @param dest Starting position in the output sequence
    @param leftSequence The left input
    @param rightSequence The right input
 */
private static <T extends Comparable<T>> void merge(T[] outputSequence,
                                                    int dest, T[] leftSequence,
                                                    T[] rightSequence) {
    int i = dest; // Index into the output sequence.
    int j = 0; // Index into the left input sequence.
    int k = 0; // Index into the right input sequence.

    // While there is data in both input sequences
    while (i < leftSequence.length && j < rightSequence.length) {
        // Find the smaller and
        // insert it into the output sequence.
        if (leftSequence[i].compareTo(rightSequence[j]) < 0) {
            outputSequence[k++] = leftSequence[i++];
        } else {
            outputSequence[k++] = rightSequence[j++];
        }
    }
    // assert: one of the sequences has more items to copy.
    // Copy remaining input from left sequence into the output.
    while (i < leftSequence.length) {
        outputSequence[k++] = leftSequence[i++];
    }
    // Copy remaining input from right sequence into output.
    while (j < rightSequence.length) {
        outputSequence[k++] = rightSequence[j++];
    }
}
```

PROGRAM STYLE

By using the postincrement operator on the index variables, you can both extract the current item from one sequence and append it to the end of the output sequence in one statement. The statement

```
outputSequence[k++] = leftSequence[i++];
```

is equivalent to the following three statements, executed in the order shown:

```
outputSequence[k] = leftSequence[i];
k++;
i++;
```

Both the single statement and the group of three statements maintain the invariant that the indexes reference the current item.

Design of Merge Sort

We can modify merging to serve as an approach to sorting a single, unsorted array as follows:

1. Split the array into halves.
2. Sort the left half.
3. Sort the right half.
4. Merge the two.

What sort algorithm should we use to do Steps 2 and 3? We can use the merge sort algorithm we are developing! The base case will be a table of size 1, which is already sorted, so there is nothing to do for the base case. We refine the algorithm next, showing its recursive steps (7. and 8.).

Refinement of Algorithm for Merge Sort

1. **if** the `tableSize is > 1`
2. Set `halfSize` to `tableSize` divided by 2.
3. Allocate a table called `leftTable` of size `halfSize`.
4. Allocate a table called `rightTable` of size `tableSize - halfSize`.
5. Copy the elements from `table[0 .. halfSize - 1]` into `leftTable`.
6. Copy the elements from `table[halfSize .. tableSize]` into `rightTable`.
7. Recursively apply the merge sort algorithm to `leftTable`.
8. Recursively apply the merge sort algorithm to `rightTable`.
9. Apply the merge method using `leftTable` and `rightTable` as the input and the original table as the output.

Trace of Merge Sort Algorithm

Figure 8.5 illustrates the merge sort. Each recursive call to method `sort` with an array argument that has more than one element splits the array argument into a left array and a right array, where each new array is approximately half the size of the array argument. We then sort each of these arrays, beginning with the left half, by recursively calling method `sort` with the left array and right array as arguments. After returning from the sort of the left array and

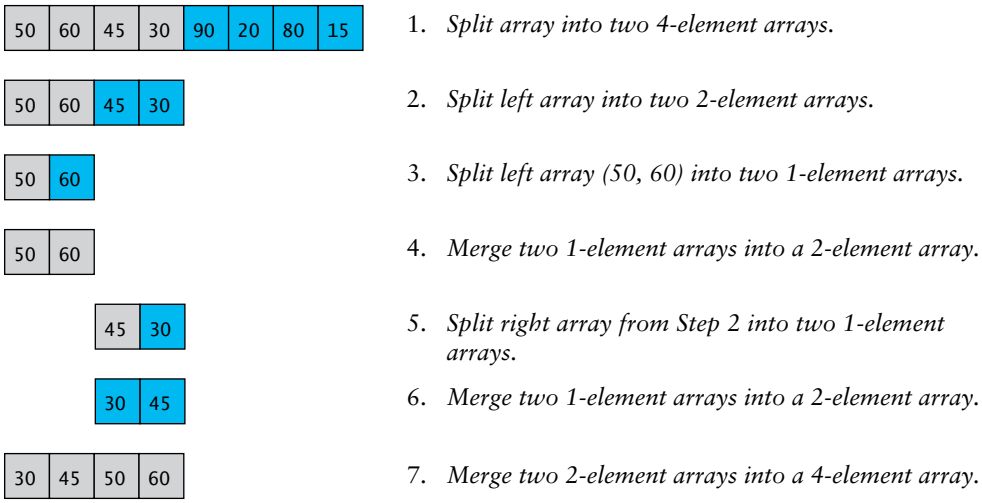
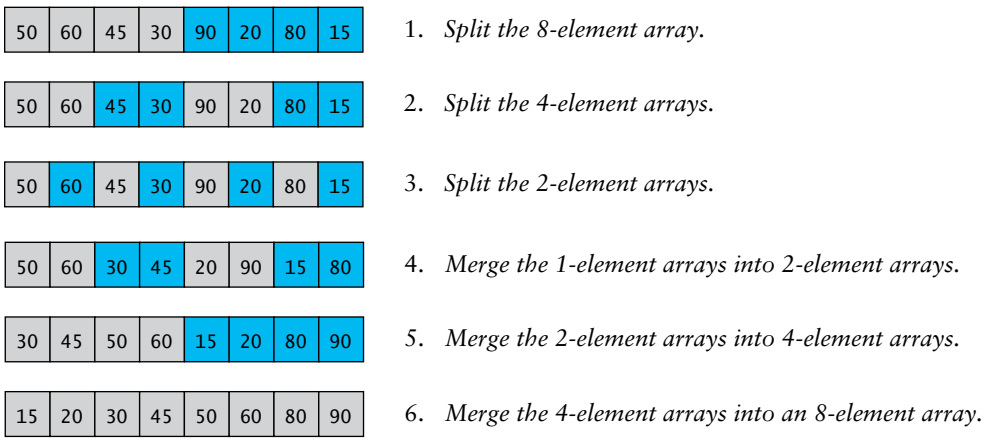


FIGURE 8.5
Trace of Merge Sort

right array at each level, we merge these halves together back into the space occupied by the array that was split. The left subarray in each recursive call (in gray) will be sorted before the processing of its corresponding right subarray (in color) begins. Lines 4 and 6 merge two one-element arrays to form a sorted two-element array. At line 7, the two sorted two-element arrays (50, 60 and 30, 45) are merged into a sorted four-element array. Next, the right subarray in color on line 1 will be sorted in the same way. When done, the sorted subarray (15, 20, 80, 90) will be merged with the sorted subarray on line 7.

Analysis of Merge Sort

In Figure 8.5, the size of the arrays being sorted decreases from 8 to 4 (line 1) to 2 (line 2) to 1 (line 3). After each pair of subarrays is sorted, the pair will be merged to form a larger sorted array. Rather than showing a time sequence of the splitting and merging operations, we summarize them as follows:



Lines 1 through 3 show the splitting operations, and lines 4 through 6 show the merge operations. Line 4 shows the two-element arrays formed by merging two-element pairs, line 5 shows the four-element arrays formed by merging two-element pairs, and line 6 shows the sorted array. Because each of these lines involves a movement of n elements from smaller-size

arrays to larger arrays, the effort to do each merge is O(n). The number of lines that require merging (three in this case) is log n because each recursive step splits the array in half. So the total effort to reconstruct the sorted array through merging is O(n log n).

Recall from our discussion of recursion that whenever a recursive method is called, a copy of the local variables is saved on the run-time stack. Thus, as we go down the recursion chain sorting the `leftTables`, a sequence of `rightTables` of size $\frac{n}{2}, \frac{n}{4}, \ldots, \frac{n}{2^k}$ is allocated. Since $\frac{n}{2} + \frac{n}{4} + \cdots + 2 + 1 = n - 1$, a total of n additional storage locations are required.

Implementation of Merge Sort

Listing 8.5 shows the `MergeSort` class.

LISTING 8.5
MergeSort.java

```
/** Implements the recursive merge sort algorithm. In this version,
    copies of the subtables are made, sorted, and then merged.
 */
public class MergeSort {
    /** Sort the array using the merge sort algorithm.
        pre: table contains Comparable objects.
        post: table is sorted.
        @param table The array to be sorted
     */
    public static <T extends Comparable<T>> void sort(T[] table) {
        // A table with one element is sorted already.
        if (table.length > 1) {
            // Split table into halves.
            int halfSize = table.length / 2;
            T[] leftTable = (E[]) new Comparable[halfSize];
            T[] rightTable = (E[]) new Comparable[table.length - halfSize];
            System.arraycopy(table, 0, leftTable, 0, halfSize);
            System.arraycopy(table, halfSize, rightTable, 0,
                             table.length - halfSize);
            // Sort the halves.
            sort(leftTable);
            sort(rightTable);

            // Merge the halves.
            merge(table, 0, leftTable, rightTable);
        }
    }
    // Insert merge method in Listing 8.4.
}
```

EXERCISES FOR SECTION 8.6

SELF-CHECK

1. Trace the execution of the merge sort on the following array, providing a figure similar to Figure 8.5.

   55 50 10 40 80 90 60 100 70 80 20 50 22

2. For the array in Question 1 above, show the value of `halfSize` and arrays `leftTable` and `rightTable` for each recursive call to method `sort` in Listing 8.5 and show the array elements after returning from each call to `merge`. How many times is `sort` called, and how many times is `merge` called?

PROGRAMMING

1. Add statements that trace the progress of method sort by displaying the array table after each merge operation. Also display the arrays referenced by `leftTable` and `rightTable`.

8.7 Timsort

Timsort was developed by Tim Peters in 2002 as the library sort algorithm for the Python language. Timsort is a modification of the merge sort algorithm that takes advantage of sorted subsets that may exist in the data being sorted. The Java 8 API replaced merge sort with Timsort as the sort algorithm for lists of objects.

Recall that merge sort recursively divides the array in half until there are two sequences of length 1. It then merges the adjacent sequences. This is depicted in Figure 8.5. As the algorithm progresses, the run-time stack holds the left-hand sequences that are waiting to be merged with the right-hand sequences. Merge sort starts with two sequences of length 1 and merges them to form a sequence of length 2. It then takes the next two sequences of length 1 and merges them. It then merges the two sequences of length 2 to form a sequence of length 4. The process then repeats building two more sequences of length 2 to create a new sequence of length 4, which is then merged with the previous sequence of length 4 to form a sequence of length 8, and so on. If the data contains subsequences that are already sorted, they are still broken down into sequences of length 1 and then merged back together.

Timsort, on the other hand, identifies sequences that are already sorted in either ascending or descending order (called a *run*). A descending sequence is in-place converted to an ascending sequence. The run is then placed onto a stack.

In merge sort, the stack contains sequences of decreasing size such that the sequence at position $(i - 2)$ is twice the length of the sequence at position $(i - 1)$ and four times the length of the sequence at position (i). Thus, the sequence at $(i - 2)$ is longer than the sum of the lengths of the sequences at positions i and $(i - 1)$, and the sequence at position $(i - 1)$ is longer than the sequence at position i.

Timsort maintains similar invariants as summarized below where i is the stack index of the new run pushed onto the stack.

1. The run at index $i - 2$ is longer than the sum of the length of the runs at indexes i and $i - 1$.
2. The run at index $i - 1$ is longer than the run at index i.

If both invariants are not satisfied, they are restored by merging adjacent runs. If there are three or more runs on the stack, the length of the run at $i - 2$ is compared to the sum of the lengths of the runs at indexes i and $i - 1$. If invariant 1 is not satisfied and the run at $i - 2$ is shorter than the run at i, then the runs at $i - 1$ and $i - 2$ are merged; otherwise, the runs at $i - 1$ and i are merged. After the merge is completed, the invariant is again checked and the process repeats until the invariant is restored. To satisfy invariant 2, the run at index $i - 1$

must be longer than the run at index *i*. If not, they are merged. When both invariants are satisfied, the next run is pushed onto the stack. After all of the runs have been processed, the stack is collapsed by repeatedly merging the top two items on the stack.

For example, consider the following array to be sorted:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
1	3	5	7	9	8	6	4	12	14	15	16	17	11	10

The stack will contain the start position of the first item in a run and its length. The run from [0] to [4] is identified and is pushed onto the stack at stack index 0.

Index	Start	Length
0	0	5

Because the next run (from [5] to [7]) is a descending run. It is in-place converted to an ascending run as shown next

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
1	3	5	7	9	4	6	8	12	14	15	16	17	11	10

and it is pushed on the stack at stack index 1 (note: the item with the highest index is at the top of the stack).

Index	Start	Length
0	0	5
1	5	3

Since the new run is shorter than the sequence before it in the stack, we look for the next run. This run from [8] to [12]. It is an ascending run and is pushed onto the stack.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
1	3	5	7	9	4	6	8	12	14	15	16	17	11	10

Index	Start	Length
0	0	5
1	5	3
2	8	5

The length (5) of the run at index 0 is shorter than the combined length (8) of the runs at 1 and 2, so invariant 1 is not satisfied. Since the run at 2 is not shorter than the run at 0, we merge the runs at 1 and 2 and collapse the stack.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
1	3	5	7	9	4	6	8	12	14	15	16	17	11	10

Index	Start	Length
0	0	5
1	5	8

Invariant 1 is not satisfied since the length of the run at stack index 0 is less than the length of the run at 1. Therefore, these two runs are merged.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
1	3	4	5	6	7	8	9	12	14	15	16	17	11	10

Index	Start	Length
0	0	13

Now the next run is identified from [13] to [14]. It is a descending run, so it is in-place converted to an ascending run and pushed onto the stack.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
1	3	4	5	6	7	8	9	12	14	15	16	17	10	11

Index	Start	Length
0	0	13
1	13	2

The run at index 0 is longer than the run at index 1, so the invariant is satisfied. Because there are no more runs, we finish by merging the two runs on the stack.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
1	3	4	5	6	7	8	9	10	11	12	14	15	16	17

This leaves us with one run and we are done. If there were more runs on the stack, the merge would continue merging the top two runs until only one run remained.

Index	Start	Length
0	0	15

Algorithm for Timsort

1. Set `lo` to zero.
2. **`while`** `lo` is less than the length of the array
3. Find the length of the run starting at `lo`.
4. If this is a decreasing run, reverse it.
5. Add this run to the stack.
6. Set `lo` to `lo` + length of the run.
7. Collapse-merge the stack to satisfy the invariants.
8. Force-merge the stack until there is only one run on it.

Algorithm to Find a Run

1. Set `hi` to `lo + 1`.
2. **if** `a[hi] < a[lo]`
3. **while** `hi < a.length and a[hi] < a[lo]`
4. increment hi

else

5. **while** `hi < a.length and a[hi] ≥ a[lo]`
6. increment hi
7. Return `hi - lo` as the length of the run.

Algorithm to Reverse a Run

1. Set `i` to the beginning of the run.
2. Set `j` to the end of the run (`i + length - 1`).
3. **while** `i < j`
4. swap `a[i]` and `a[j]`
5. increment `i`
6. decrement `j`

Algorithm for Collapse-Merge

1. **while** the stack size is > 1
2. Let `top` be the index of the top of the stack.
3. **if** `top > 1` and `stack[top-2].length ≤ stack[top-1].length + stack[top].length`
4. **if** `stack[top-2].length < stack[top].length`
5. Merge the runs at `stack[top-2]` and `stack[top-1]`.

else

6. Merge the runs at `stack[top-1]` and `stack[top]`.
7. **else if** `stack[top-1].length ≤ stack[top].length`
8. Merge the runs at `stack[top-1]` and `stack[top]`.

else

9. Exit the **while** loop.

Algorithm for Force-Merge

1. **while** the stack size is > 1
2. Merge the runs at `stack[top-1]` and `stack[top]`.

Merging Adjacent Runs

The actual merging of runs is the same as in merge sort. However, to avoid unnecessary copying of data to the temporary arrays, the start of the left run is adjusted to the first item that is greater than the first item of the right run. Likewise, the end of the right run is adjusted to be the first item that is less than the last item in the left run. These enhancements are left as programming exercises.

Performance of Timsort

In the worst case, an array with all runs of length 2, the performance would be O(n log n). The best case would be an array that is already sorted which would be O(n). The storage

requirements are O(n) since the space required to hold the temporary arrays is the same as the size of the array being sorted. Also, the stack needed to store the start index and length of each run has a length of at most log n.

Implementation of Timsort

Listing 8.6 shows the implementation of Timsort. An `ArrayList` is used for the stack because we need to access the top three items. The internal class `Run` is used to represent the runs.

LISTING 8.6
Timsort.java

```
/** Sorts an array using a simplified version of the Timsort algorithm.
 */
public class Timsort {

    /** Private inner class to hold definitions of the runs
     */
    private static class Run {
        // Data fields
        int startIndex; // start position of run
        int length;     // length of run

        public Run(int startIndex, int length) {
            this.startIndex = startIndex;
            this.length = length;
        }
    }

    /** Array of runs that are pending merging.*/
    private static List<Run> runStack;

    // Methods
    /** Sorts the array using the Timsort algorithm.
        pre: table contains Comparable objects.
        post: table is sorted.
        @param table The array to be sorted
     */
    public static <T extends Comparable<T>> void sort(T[] table) {
        runStack = new ArrayList<>(); // Create empty stack
        int nRemaining = table.length;
        if (nRemaining < 2) {
            return; // Single item array is already sorted.
        }
        int lo = 0;
        do {
           int runLength = nextRun(table, lo);
           runStack.add(new Run(lo, runLength));
           mergeCollapse(table);
           lo = lo + runLength;
           nRemaining = nRemaining - runLength;
        } while (nRemaining != 0);
        mergeForce(table);
    }

    /** Method to find the length of the next run.
        A run is a sequence of ascending items such that
        a[i] <= a[i+1] or descending items such that
        a[i] >= a[i+1]. If a descending sequence is
```

```
    found, it is turned into an ascending sequence.
    @param table The array to be sorted
    @param lo The index where the sequence starts
    @return the length of the sequence.
 */
private static <T extends Comparable<T>> int nextRun(
                                  T[] table, int lo) {
    if (lo == table.length - 1) {
        return 1;
    }
    int hi = lo + 1;
    if (table[hi - 1].compareTo(table[hi]) <= 0) {
        while (hi < table.length &&
                table[hi - 1].compareTo(table[hi]) <= 0) {
            hi++;
        }
    } else {
        while (hi < table.length &&
                table[hi - 1].compareTo(table[hi]) > 0) {
            hi++;
        }
        swapRun(table, lo, hi - 1);
    }
    return hi - lo;
}

/** Convert a descending run into an ascending run.
    @param table The array to be sorted
    @param lo The start index
    @param hi The end index
 */
private static <T extends Comparable<T>> void swapRun(
                          T[] table, int lo, int hi) {
    while (lo < hi) {
        swap(table, lo, hi);
        lo++; hi--;
    }
}

/** Swap the items in table[i] and table[j].
    @param table The array to be sorted
    @param i The index of one item
    @param j The index of the other item
 */
private static <T extends Comparable<T>> void swap(
                         T[] table, int i, int j) {
    T temp = table[i];
    table[i] = table[j];
    table[j] = temp;
}

/** Merge adjacent runs until the invariant below is satisfied.
      1. runLength[top - 2] >
         runLength[top - 1] + runLength[top]
      2. runLength[top - 1] > runLength[top]
    Called each time a new run is added to the stack.
    Invariant is true before a new run is added.
    @param table The array to be sorted
 */
 private static <T extends Comparable<T>> void mergeCollapse(T[] table) {
    while (runStack.size() > 1) {
```

```
            int top = runStack.size() - 1;
            if (top > 1 && runStack.get(top - 2).length <=
                    runStack.get(top - 1).length +
                            runStack.get(top).length) {
                if (runStack.get(top - 2).length <
                            runStack.get(top).length) {
                    // merge runs at top-2, top-1
                    mergeAt(table, top - 2);
                } else {
                    // merge runs at top-1, top
                    mergeAt(table, top - 1);
                }
            } else if (runStack.get(top - 1).length <=
                        runStack.get(top).length) {
                // merge runs at top-1, top
                mergeAt(table, top - 1);
            } else {
                break; // invariant is satisfied
            }
        }
    }

    /** Merge all remaining runs to complete the sort.
        Merge runs at top - 1 and top until only 1 run on stack
        @param table The array to be sorted
     */
    private static <T extends Comparable<T>> void
                              mergeForce(T[] table) {
        while (runStack.size() > 1) {
            int top = runStack.size() - 1;
            mergeAt(table, top - 1);
        }
    }

    /** Merges stack elements at i and i + 1
        @param table The array to be sorted
        @param i The stack index of left run to be merged
               with right run at i + 1
     */
    private static <T extends Comparable<T>> void
                               mergeAt(T[] table, int i) {
        int base1 = runStack.get(i).startIndex;
        int len1 = runStack.get(i).length;
        int base2 = runStack.get(i + 1).startIndex;
        int len2 = runStack.get(i + 1).length;
        runStack.set(i, new Run(base1, len1 + len2));
        if (i == runStack.size() - 3) {
            runStack.set(i + 1, runStack.get(i + 2));
        }
        // pop the stack
        runStack.remove(runStack.size() - 1);
        // Merge runs only if last element in left run >
        // first element in right run
        if ((table[base1 + len1 - 1]).compareTo(table[base2]) > 0) {
            T[] run1 = (T[]) (new Comparable[len1]);
            T[] run2 = (T[]) (new Comparable[len2]);
            System.arraycopy(table, base1, run1, 0, len1);
            System.arraycopy(table, base2, run2, 0, len2);
            merge(table, base1, run1, run2);
        }
    }
    // Insert merge method in Listing 8.4
}
```

EXERCISES FOR SECTION 8.7

SELF-CHECK

1. Simulate the execution of Timsort on each array below. Show the array and the stack after each execution of the `while` loop in method `mergeCollapse`. For the array in part a. below, after the first two runs are pushed onto the stack, the stack and array should appear as:

```
index    start    length
0          0        2
1          2        3

{4, 10, 9, 11, 18, 11, 1, 7, 3, 10, 10, 14}
```

   The array above shows the `String` result of `Arrays.toString(table)`. The arrays to be sorted follow.

   a. `{4, 10, 18, 11, 9, 11, 1, 7, 3, 10, 10, 14}`

   b. `{1, 9, 13, 13, 15, 15, 20, 22, 22, 10, 18, 19, 1, 9, 16}`

2. When merging adjacent runs, performance can be improved if the start of the left run is adjusted to be the first item that is greater than the first item of the right run. Likewise, the end of the right run can be adjusted to be the first item that is less than the last item in the left run. Show the subarrays to be merged for the runs below:

   left run is {10, 12, 14, 18, 19}, right run is {13, 15, 16, 17, 20, 23}

PROGRAMMING

1. Insert statements to trace the execution of `mergeCollapse` displaying the array and stack in a form similar to what is shown in Self-Check Exercise 1 above.
2. Modify method `mergeAt` to incorporate the refinement shown in Exercise 2 above.

8.8 Heapsort

The merge sort algorithm has the virtue that its time is O(n log n), but it still requires, at least temporarily, n extra storage locations. This next algorithm can be implemented without requiring any additional storage. It uses a heap to store the array and so is called *heapsort*.

First Version of a Heapsort Algorithm

We introduced the heap in Section 6.6. When used as a priority queue, a heap is a data structure that maintains the smallest value at the top. The following algorithm first places an array's data into a heap. Then it removes each heap item (an O(log n) process) and moves it back into the array.

Heapsort Algorithm: First Version

1. Insert each value from the array to be sorted into a priority queue (heap).
2. Set `i` to 0.

3. **while** the priority queue is not empty
4. Remove an item from the queue and insert it back into the array at position i.
5. Increment `i`.

Although this algorithm can be shown to be O(n log n), it does require n extra storage locations (the array and heap are both size n). We address this problem next.

Revising the Heapsort Algorithm

In the heaps we have used so far, each parent node value was less than the values of its children. We can also build the heap so that each parent is larger than its children. Figure 8.6 shows an example of such a heap.

Once we have such a heap, we can remove one item at a time from the heap. The item removed is always the top element, and it will end up at the bottom of the heap. When we reheap, we move the larger of a node's two children up the heap, instead of the smaller, so the next largest item is then at the top of the heap. Figure 8.7 shows the heap after we have removed one item, and Figure 8.8 shows the heap after we have removed two items. In both figures, the items in bold have been removed from the heap. As we continue to remove items from the heap, the heap size shrinks as the number of the removed items increases. Figure 8.9 shows the heap after we have emptied it.

If we implement the heap using an array, each element removed will be placed at the end of the array but in front of the elements that were removed earlier. After we remove the last element, the array will be sorted. We illustrate this next.

Figure 8.10 shows the array representation of the original heap. As before, the root, 89, is at position 0. The root's two children, 76 and 74, are at positions 1 and 2. For a node at position p, the left child is at $2p + 1$ and the right child is at $2p + 2$. A node at position c can find its parent at $(c - 1)/2$.

Figure 8.11 shows the array representation of the heaps in Figures 8.7 through 8.9. The items in dark color have been removed from the heap and are sorted. Each time an item is removed, the heap part of the array decreases by one element and the sorted part of the array increases

FIGURE 8.6
Example of a Heap with Largest Value in Root

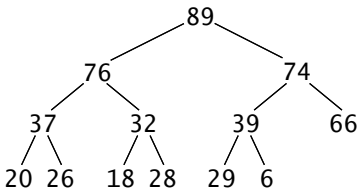


FIGURE 8.7
Heap after Removal of Largest Item

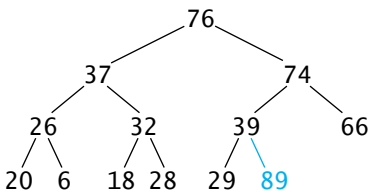


FIGURE 8.8
Heap after Removal of Two Largest Items

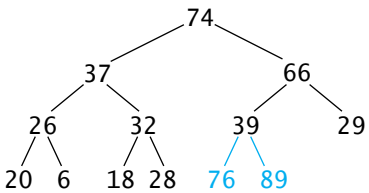


FIGURE 8.9
Heap after Removal of All Its Items

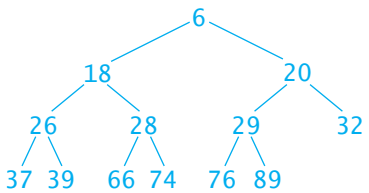


FIGURE 8.10
Internal Representation of the Heap Shown in Figure 8.6

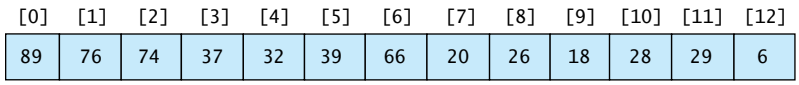
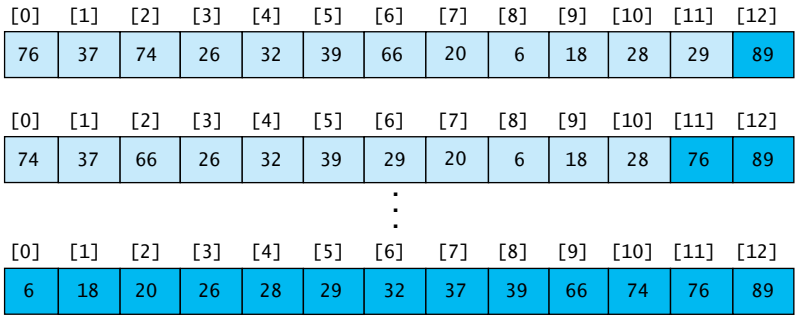


FIGURE 8.11
Internal Representation of the Heaps Shown in Figures 8.7 through 8.9



by one element. In the array at the bottom of Figure 8.11, all items have been removed from the heap and the array is sorted.

From our foregoing observations, we can sort the array that represents a heap in the following way.

Algorithm for In-Place Heapsort

1. Build a heap by rearranging the elements in an unsorted array.
2. **while** the heap is not empty
3. Remove the first item from the heap by swapping it with the last item in the heap and restoring the heap property.

Each time through the loop (Steps 2 and 3), the largest item remaining in the heap is placed at the end of the heap, just before the previously removed items. Thus, when the loop terminates, the items in the array are sorted. In Section 6.6, we discussed how to remove an item from a heap and restore the heap property. We also implemented a `remove` method for a heap in an `ArrayList`.

Algorithm to Build a Heap

Step 1 of the algorithm builds a heap. If we start with an array `table` of length `table.length`, we can consider the first item (index 0) to be a heap of one item. We now consider the general case where the items in array `table` from 0 through `n - 1` form a heap; the items from `n` through `table.length - 1` are not in the heap. As each is inserted, we must "reheap" to restore the heap property.

Refinement of Step 1 for In-Place Heapsort

1.1 **while** `n` is less than `table.length`
1.2 Increment `n` by 1. This inserts a new item into the heap.
1.3 Restore the heap property.

Analysis of Revised Heapsort Algorithm

From our knowledge of binary trees, we know that a heap of size n has $\log n$ levels. Building a heap requires finding the correct location for an item in a heap with $\log n$ levels. Because we

have n items to insert and each insert (or remove) is $O(\log n)$, the `buildHeap` operation is $O(n \log n)$. Similarly, we have n items to remove from the heap, so that is also $O(n \log n)$. Because we are storing the heap in the original array, no extra storage is required.

Implementation of Heapsort

Listing 8.7 shows the `HeapSort` class. The `sort` method merely calls the `buildHeap` method followed by the `shrinkHeap` method, which is based on the `remove` method shown in Section 6.6. Method `swap` swaps the items in the table.

LISTING 8.7
HeapSort.java

```
/** Implementation of the heapsort algorithm. */
public class HeapSort {
    /** Sort the array using heapsort algorithm.
        pre: table contains Comparable items.
        post: table is sorted.
        @param table The array to be sorted
     */
    public static <T extends Comparable<T>> void sort(T[] table) {
        buildHeap(table);
        shrinkHeap(table);
    }

    /** buildHeap transforms the table into a heap.
        pre: The array contains at least one item.
        post: All items in the array are in heap order.
        @param table The array to be transformed into a heap
     */
    private static <T extends Comparable<T>> void buildHeap(T[] table) {
        int n = 1;
        // Invariant: table[0 . . n - 1] is a heap.
        while (n < table.length) {
            n++; // Add a new item to the heap and reheap.
            int child = n - 1;
            int parent = (child - 1) / 2; // Find parent.
            while (parent >= 0
                   && table[parent].compareTo(table[Child]) < 0) {
                swap(table, parent, child);
                child = parent;
                parent = (child - 1) / 2;
            }
        }
    }

    /** shrinkHeap transforms a heap into a sorted array.
        pre: All items in the array are in heap order.
        post: The array is sorted.
        @param table The array to be sorted
     */
    private static <T extends Comparable<T>> void shrinkHeap(T[] table) {
        int n = table.length;
        // Invariant: table[0 . . n - 1] forms a heap.
        // table[n . . table.length - 1] is sorted.
        while (n > 0) {
            n-- ;
            swap(table, 0, n);
            // table[1 . . n - 1] form a heap.
            // table[n . . table.length - 1] is sorted.
```

```
                int parent = 0;
                while (true) {
                    int leftChild = 2 * parent + 1;
                    if (leftChild >= n) {
                        break;       // No more children.
                    }
                    int rightChild = leftChild + 1;
                    // Find the larger of the two children.
                    int maxChild = leftChild;
                    if (rightChild < n
                        && table[leftChild].compareTo(table[rightChild]) < 0) {
                        maxChild = rightChild;
                    }
                    // If the parent is smaller than the larger child,
                    if (table[parent].compareTo(table[maxChild]) < 0) {
                        // Swap the parent and child.
                        swap(table, parent, maxChild);
                        // Continue at the child level.
                        parent = maxChild;
                    } else { // Heap property is restored.
                        break; // Exit the loop.
                    }
                }
            }
        }

        /** Swap the items in table[i] and table[j].
            @param table The array that contains the items
            @param i The index of one item
            @param j The index of the other item
         */
        private static <T extends Comparable<T>> void swap(T[] table,
                                                           int i, int j) {
            T temp = table[i];
            table[i] = table[j];
            table[j] = temp;
        }
    }
```

EXERCISES FOR SECTION 8.8

SELF-CHECK

1. Build the heap from the numbers in the following list. How many exchanges were required? How many comparisons?

   55 50 10 40 80 90 60 100 70 80 20 50 22

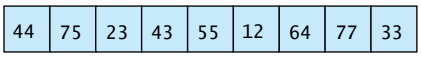
2. Shrink the heap from Question 1 to create the array in sorted order. How many exchanges were required? How many comparisons?

8.9 Quicksort

The next algorithm we will study is called *quicksort*. Developed by C. A. R. Hoare in 1962, it works in the following way: given an array with subscripts `first . . last` to sort, quicksort rearranges this array into two parts so that all the elements in the left subarray are less than or equal to a specified value (called the *pivot)* and all the elements in the right subarray are

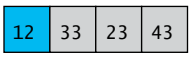
greater than the pivot. The pivot is placed between the two parts. Thus, all of the elements on the left of the pivot value are smaller than all elements on the right of the pivot value, so the pivot value is in its correct position. By repeating this process on the two halves, the whole array eventually becomes sorted.

As an example of this process, let's sort the following array:

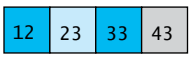


We will assume that the first array element (44) is arbitrarily selected as the pivot value. A possible result of rearranging, or *partitioning,* the element values follows:

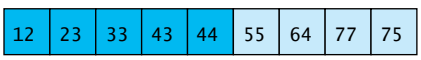
After the partitioning process, the pivot value, **44**, is at its correct position. All values less than **44** are in the left subarray (light blue shading), and all values larger than **44** are in the right subarray (gray shading), as desired. The next step would be to apply quicksort recursively to the two subarrays on either side of the pivot value, beginning with the left subarray (12, 33, 23, 43). Here is the result when 12 is the pivot value:



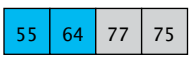
The pivot value is in the first position. Because the left subarray does not exist, the right subarray (33, 23, 43) is sorted next, resulting in the following situation:



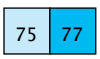
The pivot value 33 is in its correct place, and the left subarray (23) and right subarray (43) have single elements, so they are sorted. At this point, we are finished sorting the left part of the original subarray, and quicksort is applied to the right subarray (55, 64, 77, 75). In the following array, all the elements that have been placed in their proper position are in dark blue.



If we use 55 for the pivot, its left subarray will be empty after the partitioning process and the right subarray 64, 77, 75 will be sorted next. If 64 is the pivot, the situation will be as follows, and we sort the right subarray (77, 75) next.



If 77 is the pivot and we move it where it belongs, we end up with the following array. Because the left subarray (75) has a single element, it is sorted and we are done.



Algorithm for Quicksort

The algorithm for quicksort follows. We will describe how to do the partitioning later. We assume that the indexes `first` and `last` are the endpoints of the array being sorted and that the index of the pivot after partitioning is `pivIndex`.

Algorithm for Quicksort

1. `if first < last` then
2. Partition the elements in the subarray `first .. last` so that the pivot value is in its correct place (subscript `pivIndex`).
3. Recursively apply quicksort to the subarray `first .. pivIndex - 1`.
4. Recursively apply quicksort to the subarray `pivIndex + 1 .. last`.

Analysis of Quicksort

If the pivot value is a random value selected from the current subarray, then statistically it is expected that half of the items in the subarray will be less than the pivot and half will be greater than the pivot. If both subarrays always have the same number of elements (the best case), there will be log n levels of recursion. At each level, the partitioning process involves moving every element into its correct partition, so quicksort is O(n log n), just like merge sort.

But what if the split is not 50–50? Let us consider the case where each split is 90–10. Instead of a 100-element array being split into two 50-element arrays, there will be one array with 90 elements and one with just 10. The 90-element array may be split 50–50, or it may also be split 90–10. In the latter case, there would be one array with 81 elements and one with just 9 elements. Generally, for random input, the splits will not be exactly 50–50, but neither will they all be 90–10. An exact analysis is difficult and beyond the scope of this book, but the running time will be bound by a constant × n log n.

There is one situation, however, where quicksort gives very poor behavior. If, each time we partition the array, we end up with a subarray that is empty, the other subarray will have one less element than the one just split (only the pivot value will be removed). Therefore, we will have n levels of recursive calls (instead of log n), and the algorithm will be $\mathbf{O}(n^2)$. Because of the overhead of recursive method calls (versus iteration), quicksort will take longer and require more extra storage on the run-time stack than any of the earlier quadratic algorithms. We will discuss a way to handle this situation later.

Implementation of Quicksort

Listing 8.8 shows the `QuickSort` class. The public method `sort` calls the recursive `quickSort` method, giving it the bounds of the `table` as the initial values of `first` and `last`. The two recursive calls in `quickSort` will cause the procedure to be applied to the subarrays that are separated by the value at `pivIndex`. If any subarray contains just one element (or zero elements), an immediate return will occur.

LISTING 8.8

QuickSort.java

```
/** Implements the quicksort algorithm. */
public class QuickSort {
    /** Sort the table using the quicksort algorithm.
        pre: table contains Comparable objects.
        post: table is sorted.
        @param table The array to be sorted
     */
    public static <T extends Comparable<T>> void sort(T[] table) {
        // Sort the whole table.
        quickSort(table, 0, table.length - 1);
    }
```

```
    /** Sort a part of the table using the quicksort algorithm.
        post: The part of table from first through last is sorted.
        @param table The array to be sorted
        @param first The index of the low bound
        @param last The index of the high bound
     */
    private static <T extends Comparable<T>> void quickSort(T[] table,
                                                            int first, int last) {
        if (first < last) { // There is data to be sorted.
            // Partition the table.
            int pivIndex = partition(table, first, last);
            // Sort the left half.
            quickSort(table, first, pivIndex - 1);
            // Sort the right half.
            quickSort(table, pivIndex + 1, last);
        }
    }
    // Insert partition method. See Listing 8.9 or 8.10
    . . .
}
```

Algorithm for Partitioning

The `partition` method selects the pivot and performs the partitioning operation. When we are selecting the pivot, it does not really matter which element is the pivot value (if the arrays are randomly ordered to begin with). For simplicity, we chose the element with subscript `first`. We then begin searching for the first value at the left end of the subarray that is greater than the pivot value. When we find it, we search for the first value at the right end of the subarray that is less than or equal to the pivot value. These two values are exchanged, and we repeat the search and exchange operations. This is illustrated in Figure 8.12, where `up` points to the first value greater than the pivot and `down` points to the first value less than or equal to the pivot value. The elements less than the pivot are in light color, and the elements greater than the pivot are in gray.

The value 75 is the first value at the left end of the array that is larger than 44, and 33 is the first value at the right end that is less than or equal to 44, so these two values are exchanged. The indexes `up` and `down` are advanced again, as shown in Figure 8.13.

The value 55 is the next value at the left end that is larger than 44, and 12 is the next value at the right end that is less than or equal to 44, so these two values are exchanged, and `up` and `down` are advanced again, as shown in Figure 8.14.

After the second exchange, the first five array elements contain the pivot value and all values less than or equal to the pivot; the last four elements contain all values larger than the

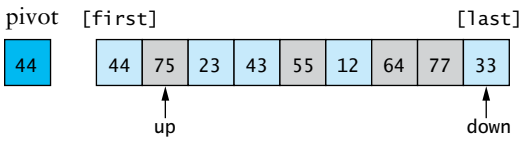


FIGURE 8.12
Locating First Values to Exchange

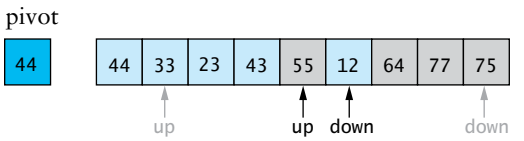


FIGURE 8.13
Array after the First Exchange

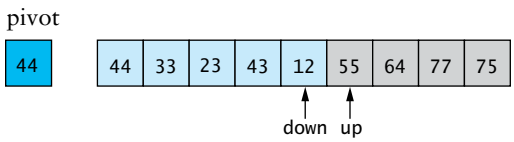


FIGURE 8.14
Array after the Second Exchange

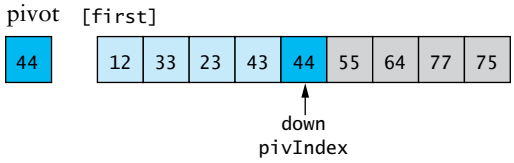


FIGURE 8.15
Array after the Pivot Is Inserted

pivot. The value 55 is selected once again by `up` as the next element larger than the pivot; 12 is selected by `down` as the next element less than or equal to the pivot. Since `up` has now "passed" `down`, these values are not exchanged. Instead, the pivot value (subscript `first`) and the value at position `down` are exchanged. This puts the pivot value in its proper position (the new subscript is `down`) as shown in Figure 8.15.

The partition process is now complete, and the value of `down` is returned to the pivot index `pivIndex`. Method `quickSort` will be called recursively to sort the left subarray and the right subarray. The algorithm for `partition` follows:

Algorithm for `partition` Method

1. Define the pivot value as the contents of `table[first]`.
2. Initialize `up` to `first` and `down` to `last`.
3. **`do`**
4. Increment `up` until `up` selects the first element greater than the pivot value or `up` has reached `last`.
5. Decrement `down` until `down` selects the first element less than or equal to the pivot value or `down` has reached `first`.
6. **`if`** `up < down` then
7. Exchange `table[up]` and `table[down]`.
8. **`while`** `up < down`
9. Exchange `table[first]` and `table[down]`.
10. Return the value of `down` to `pivIndex`.

Implementation of `partition`

The code for `partition` is shown in Listing 8.9. The **`while`** statement:

```
while ((up < last) && (pivot.compareTo(table[up]) >= 0)) {
    up++;
}
```

advances the index `up` until it is equal to `last` or until it references an item in `table` that is greater than the pivot value. Similarly, the **`while`** statement:

```
while (pivot.compareTo(table[down]) < 0)) {
    down--;
}
```

moves the index `down` until it references an item in `table` that is less than or equal to the pivot value. The **`do-while`** condition

```
(up < down)
```

ensures that the partitioning process will continue while `up` is to the left of `down`.