Chapter 16 Advanced Data Structures

CHAPTER GOALS

- To learn about the set and map data types
- To understand the implementation of hash tables
- To be able to program hash functions
- To learn about binary trees
- To be able to use tree sets and tree maps
- To become familiar with the heap data structure
- To learn how to implement the priority queue data type
- To understand how to use heaps for sorting

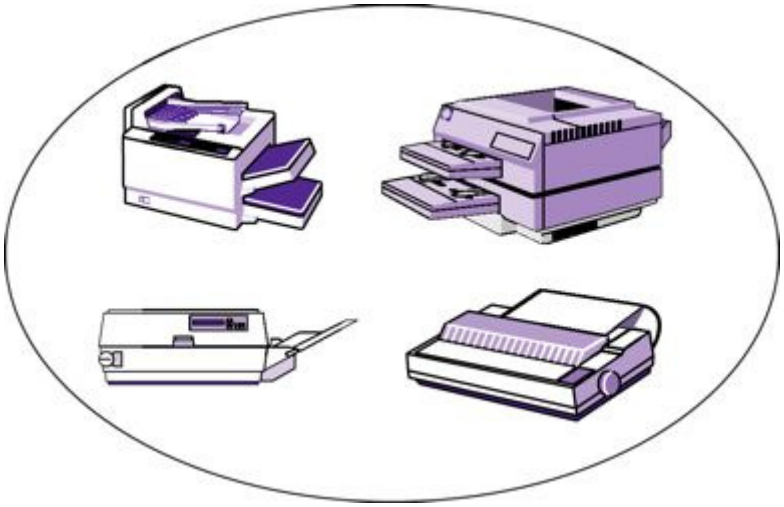
In this chapter we study data structures that are more complex than arrays or lists. These data structures take control of organizing their elements, rather than keeping them in a fixed position. In return, they can offer better performance for adding, removing, and finding elements.

You will learn about the abstract set and map data types and the implementations that the standard library offers for these abstract types. You will see how two completely different implementations—hash tables and trees—can be used to implement these abstract types efficiently.

16.1 Sets

In the preceding chapter you encountered two important data structures: arrays and lists. Both have one characteristic in common: These data structures keep the elements in the same order in which you inserted them. However, in many applications, you don't really care about the order of the elements in a collection. For example, a server may keep a collection of objects representing available printers (see Figure 1). The order of the objects doesn't really matter.

Figure 1



A Set of Printers

In mathematics, such an unordered collection is called a *set*. You have probably learned some set theory in a course in mathematics, and you may know that sets are a fundamental mathematical notion.

> A set is an unordered collection of distinct elements. Elements can be added, located, and removed.

But what does that mean for data structures? If the data structure is no longer responsible for remembering the order of element insertion, can it give us better performance for some of its operations? It turns out that it can indeed, as you will see later in this chapter.

Let's list the fundamental operations on a set:

- Adding an element
- Removing an element
- Containment testing (does the set contain a given object?)
- Listing all elements (in arbitrary order)

In mathematics, a set rejects duplicates. If an object is already in the set, an attempt to add it again is ignored. That's useful in many programming situations as well. For example, if we keep a set of available printers, each printer should occur at most once in the set. Thus, we will interpret the `add` and `remove` operations of sets just as we do in mathematics: Adding an element has no effect if the element is already in the set, and attempting to remove an element that isn't in the set is silently ignored.

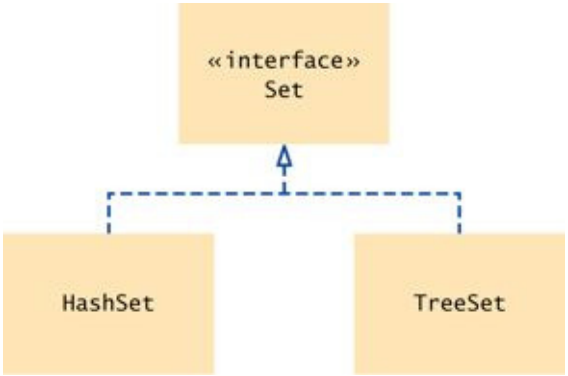
Sets don't have duplicates. Adding a duplicate of an element that is already present is silently ignored.

Of course, we could use a linked list to implement a set. But adding, removing, and containment testing would be relatively slow, because they all have to do a linear search through the list. (Adding requires a search through the list to make sure that we don't add a duplicate.) As you will see later in this chapter, there are data structures that can handle these operations much more quickly.

In fact, there are two different data structures for this purpose, called *hash tables* and *trees*. The standard Java library provides set implementations based on both data structures, called `HashSet` and `TreeSet`. Both of these data structures implement the `Set` interface (see Figure 2).

The `HashSet` and `TreeSet` classes both implement the `Set` interface.

Figure 2



Set Classes and Interfaces in the Standard Library

You will see later in this chapter when it is better to choose a hash set over a tree set. For now, let's look at an example where we choose a hash set. To keep the example simple, we'll store only strings, not `Printer` objects.

```
Set<String> names = new HashSet<String>();
```

Note that we store the reference to the `HashSet<String>` object in a `Set<String>` variable. After you construct the collection object, the implementation no longer matters; only the interface is important.

Adding and removing set elements is straightforward:

```
names.add("Romeo");
names.remove("Juliet");
```

The `contains` method tests whether an element is contained in the set:

```
if (names.contains("Juliet")) . . .
```

Finally, to list all elements in the set, get an iterator. As with list iterators, you use the `next` and `hasNext` methods to step through the set.

```
Iterator<String> iter = names.iterator();
while (iter.hasNext())
{
   String name = iter.next();
   Do something with name
}
```

Or, as with arrays and lists, you can use the "for each" loop instead of explicitly using an iterator:

```
for (String name : names)
{
   Do something with name
}
```

Note that the elements are *not* visited in the order in which you inserted them. Instead, they are visited in the order in which the `HashSet` keeps them for rapid execution of its methods.

An iterator visits all elements in a set.

A set iterator does not visit the elements in the order in which you inserted them. The set implementation rearranges the elements so that it can locate them quickly.

There is an important difference between the `Iterator` that you obtain from a set and the `ListIterator` that a list yields. The `ListIterator` has an `add` method to add an element at the list iterator position. The `Iterator` interface has no such method. It makes no sense to add an element at a particular position in a set, because the set can order the elements any way it likes. Thus, you always add elements directly to a set, never to an iterator of the set.

You cannot add an element to a set at an iterator position.

However, you can remove a set element at an iterator position, just as you do with list iterators.

Also, the `Iterator` interface has no `previous` method to go backwards through the elements. Because the elements are not ordered, it is not meaningful to distinguish between “going forward“ and “going backward”.

The following test program allows you to add and remove set elements. After each command, it prints out the current contents of the set. When you run this program, try adding strings that are already contained in the set and removing strings that aren't present in the set.

702

703

ch16/set/SetDemo.java

```
1  import java.util.HashSet;
2  import java.util.Scanner;
3  import java.util.Set;
4
5  /**
6     This program demonstrates a set of strings. The user
7     can add and remove strings.
8  */
9  public class SetDemo
10 {
11    public static void main(String[] args)
```

```
12      {
13         Set<String> names = new
HashSet<String>();
14         Scanner in = new Scanner(System.in);
15
16         boolean done = false;
17         while (!done)
18         {
19            System.out.print("Add name, Q when
done: ");
20            String input = in.next();
21            if (input.equalsIgnoreCase("Q"))
22               done = true;
23            else
24            {
25               names.add(input);
26               print(names);
27            }
28         }
29
30         done = false;
31         while (!done)
32         {
33            System.out.print("Remove name, Q when
done: ");
34            String input = in.next();
35            if (input.equalsIgnoreCase("Q"))
36               done = true;
37            else
38            {
39               names.remove(input);
40               print(names);
41            }
42         }
43      }
44
45      /**
46         Prints the contents of a set of strings.
47         @param s a set of strings
48      */
49      private static void print(Set<String> s)
50      {
```

```
51          System.out.print("{ ");
52          for (String element : s)
53          {                                                    703
54             System.out.print(element);                        704
55             System.out.print(" ");
56          }
57          System.out.println("}");
58       }
59    }
```

Output

```
Add name, Q when done: Dick
{ Dick }
Add name, Q when done: Tom
{ Tom Dick }
Add name, Q when done: Harry
{ Harry Tom Dick }
Add name, Q when done: Tom
{ Harry Tom Dick }
Add name, Q when done: Q
Remove name, Q when done: Tom
{ Harry Dick }
Remove name, Q when done: Jerry
{ Harry Dick }
Remove name, Q when done: Q
```

SELF CHECK

1. Arrays and lists remember the order in which you added elements; sets do not. Why would you want to use a set instead of an array or list?
2. Why are set iterators different from list iterators?

QUALITY TIP 16.1 Use Interface References to Manipulate Data Structures

It is considered good style to store a reference to a `HashSet` or `TreeSet` in a variable of type `Set`.

```
Set<String> names = new HashSet<String>();
```

This way, you have to change only one line if you decide to use a `TreeSet` instead.

Also, methods that operate on sets should specify parameters of type `Set`:

```
public static void print(Set<String> s)
```

Then the method can be used for all set implementations.

In theory, we should make the same recommendation for linked lists, namely to save `LinkedList` references in variables of type `List`. However, in the Java library, the `List` interface is common to both the `ArrayList` and the `LinkedList` class. In particular, it has `get` and `set` methods for random access, even though these methods are very inefficient for linked lists. You can't write efficient code if you don't know whether random access is efficient or not. This is plainly a serious design error in the standard library, and I cannot recommend using the `List` interface for that reason. (To see just how embarrassing that error is, have a look at the source code for the `binarySearch` method of the `Collections` class. That method takes a `List` parameter, but binary search makes no sense for a linked list. The code then clumsily tries to discover whether the list is a linked list, and then switches to a linear search!)

The `Set` interface and the `Map` interface, which you will see in the next section, are well-designed, and you should use them.

16.2 Maps

A map is a data type that keeps associations between *keys* and *values*. Figure 3 gives a typical example: a map that associates names with colors. This map might describe the favorite colors of various people.

A map keeps associations between key and value objects.

Mathematically speaking, a map is a function from one set, the *key set*, to another set, the *value set*. Every key in the map has a unique value, but a value may be associated with several keys.

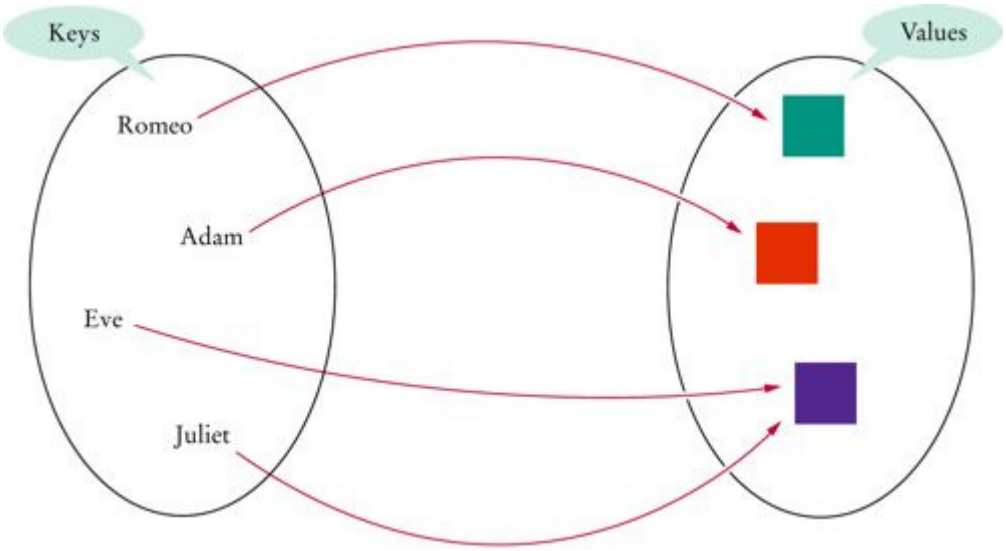
Just as there are two kinds of set implementations, the Java library has two implementations for maps: `HashMap` and `TreeMap`. Both of them implement the `Map` interface (see Figure 4).

The `HashMap` and `TreeMap` classes both implement the `Map` interface.

After constructing a `HashMap` or `TreeMap`, you should store the reference to the map object in a `Map` reference:

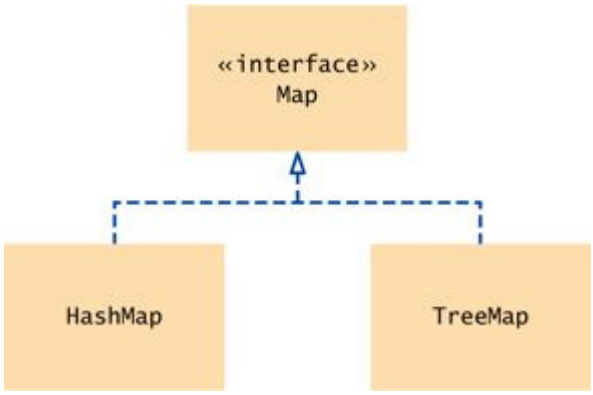
```
Map<String, Color> favoriteColors = new
HashMap<String, Color>();
```

Figure 3



A Map

Figure 4



Map Classes and Interfaces in the Standard Library

Use the `put` method to add an association:

```
favoriteColors.put("Juliet", Color.PINK);
```

You can change the value of an existing association, simply by calling `put` again:

```
favoriteColors.put("Juliet", Color.RED);
```

The `get` method returns the value associated with a key.

```
Color julietsFavoriteColor =
favoriteColors.get("Juliet");
```

If you ask for a key that isn't associated with any values, then the `get` method returns `null`.

To remove a key and its associated value, use the `remove` method:

```
favoriteColors.remove("Juliet");
```

Sometimes you want to enumerate all keys in a map. The `keySet` method yields the set of keys. You can then ask the key set for an iterator and get all keys. From each key, you can find the associated value with the `get` method. Thus, the following instructions print all key/value pairs in a map `m`:

```
Set<String> keySet = m.keySet();
for (String key : keySet)
```

```
{
   Color value = m.get(key);
   System.out.println(key + "->" + value);
}
```

The following sample program shows a map in action.

To find all keys and values in a map, iterate through the key set and find the values that correspond to the keys.

ch16/map/MapDemo.java

```
1  import java.awt.Color;
2  import java.util.HashMap;
3  import java.util.Map;
4  import java.util.Set;
5
6  /**
7     This program demonstrates a map that maps names to colors.
8  */
9  public class MapDemo
10 {
11    public static void main(String[] args)
12    {
13       Map<String, Color> favoriteColors
14             = new HashMap<String, Color>();
15       favoriteColors.put("Juliet", Color.PINK);
16       favoriteColors.put("Romeo", Color.GREEN);
17       favoriteColors.put("Adam", Color.BLUE);
18       favoriteColors.put("Eve", Color.PINK);
19
20       Set<String> keySet =
favoriteColors.keySet();
21       for (String key : keySet)
22       {
23          Color value = favoriteColors.get(key);
24          System.out.println(key + "->" +
value);
25       }
26    }
27 }
```

Output

```
Romeo->java.awt.Color[r=0,g=255,b=0]
Eve->java.awt.Color[r=255,g=175,b=175]
Adam->java.awt.Color[r=0,g=0,b=255]
Juliet->java.awt.Color[r=255,g=175,b=175]
```

SELF CHECK

3. What is the difference between a set and a map?
4. Why is the collection of the keys of a map a set?

16.3 Hash Tables

In this section, you will see how the technique of *hashing* can be used to find elements in a data structure quickly, without making a linear search through all elements. Hashing gives rise to the *hash table*, which can be used to implement sets and maps.

A *hash function* is a function that computes an integer value, the *hash code*, from an object, in such a way that different objects are likely to yield different hash codes. The `Object` class has a `hashCode` method that other classes need to redefine. The call

```
int h = x.hashCode();
```

computes the hash code of the object `x`.

A hash function computes an integer value from an object.

Table 1 Sample Strings and Their Hash Codes

String	Hash Code
"Adam"	2035631
"Eve"	70068
"Harry"	69496448
"Jim"	74478
"Joe"	74656
"Juliet"	–2065036585
"Katherine"	2079199209
"Sue"	83491

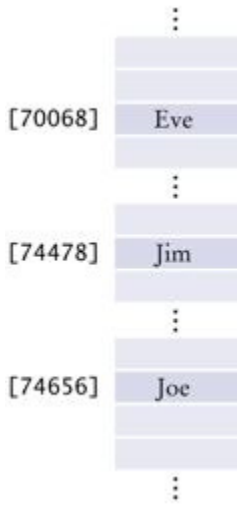
It is possible for two or more distinct objects to have the same hash code; this is called a *collision*. A good hash function minimizes collisions. For example, the `String` class defines a hash function for strings that does a good job of producing different integer values for different strings. Table 1 shows some examples of strings and their hash codes. You will see in Section 16.4 how these values are obtained.

A good hash function minimizes *collisions*—identical hash codes for different objects.

Section 16.4 explains how you should redefine the `hashCode` method for other classes.

A hash code is used as an array index into a hash table. In the simplest implementation of a hash table, you could make an array and insert each object at the location of its hash code (see Figure 5).

Figure 5



A Simplistic Implementation of a Hash Table

708

709

Then it is a very simple matter to find out whether an object is already present in the set or not. Compute its hash code and check whether the array position with that hash code is already occupied. This doesn't require a search through the entire array!

However, there are two problems with this simplistic approach. First, it is not possible to allocate an array that is large enough to hold all possible integer index positions. Therefore, we must pick an array of some reasonable size and then reduce the hash code to fall inside the array:

```
int h = x.hashCode();
if (h < 0) h = -h;
h = h % size;
```

Furthermore, it is possible that two different objects have the same hash code. After reducing the hash code modulo a smaller array size, it becomes even more likely that several objects will collide and need to share a position in the array.

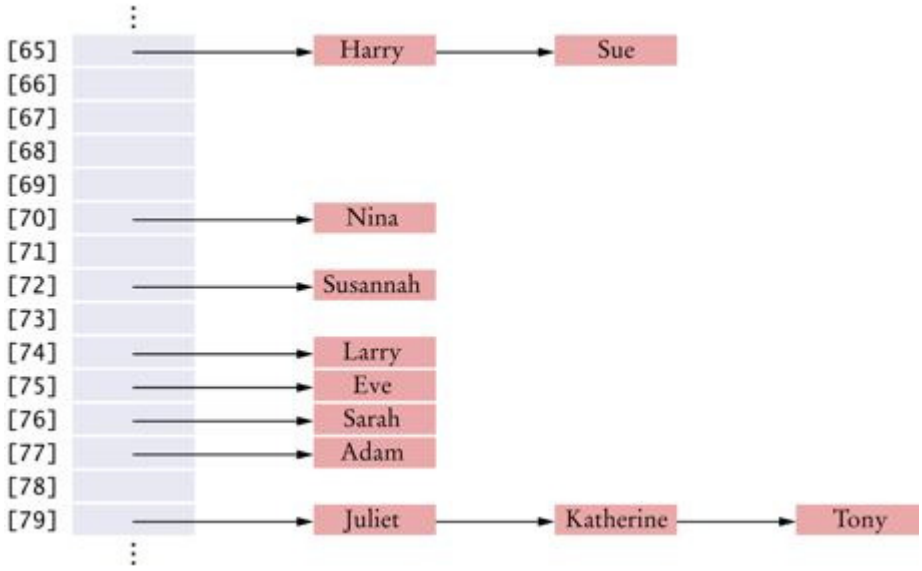
To store multiple objects in the same array position, use short node sequences for the elements with the same hash code (see Figure 6). These node sequences are called *buckets*.

A hash table can be implemented as an array of *buckets*—sequences of nodes that hold elements with the same hash code.

Now the algorithm for finding an object `x` in a hash table is quite simple.

1. Compute the hash code and reduce it modulo the table size. This gives an index `h` into the hash table.
2. Iterate through the elements of the bucket at position `h`. For each element of the bucket, check whether it is equal to `x`.
3. If a match is found among the elements of that bucket, then `x` is in the set. Otherwise, it is not.

Figure 6



A Hash Table with Buckets to Store Elements with the Same Hash Code

709

710

In the best case, in which there are no collisions, all buckets either are empty or have a single element. Then checking for containment takes constant or $O(1)$ time.

If there are no or only a few collisions, then adding, locating, and removing hash table elements takes constant or *O*(1) time.

More generally, for this algorithm to be effective, the bucket sizes must be small. If the table length is small, then collisions are unavoidable, and each bucket will get quite full. Then the linear search through a bucket is time-consuming. In the worst case, where all elements end up in the same bucket, a hash table degenerates into a linked list!

In order to reduce the chance for collisions, you should make a hash table somewhat larger than the number of elements that you expect to insert. An excess capacity of about 30 percent is typically recommended. According to some researchers, the hash table size should be chosen to be a prime number to minimize the number of collisions.

The table size should be a prime number, larger than the expected number of elements.

Adding an element is a simple extension of the algorithm for finding an object. First compute the hash code to locate the bucket in which the element should be inserted. Try finding the object in that bucket. If it is already present, do nothing. Otherwise, insert it.

Removing an element is equally simple. First compute the hash code to locate the bucket in which the element should be inserted. Try finding the object in that bucket. If it is present, remove it. Otherwise, do nothing.

As long as there are few collisions, an element can also be added or removed in constant or *O*(1) time.

At the end of this section you will find the code for a simple implementation of a hash set. That implementation takes advantage of the `AbstractSet` class, which already implements most of the methods of the `Set` interface.

In this implementation you must specify the size of the hash table. In the standard library, you don't need to supply a table size. If the hash table gets too full, a new table of twice the size is created, and all elements are inserted into the new table.

ch16/hashtable/HashSet.java

```
 1  import java.util.AbstractSet;
 2  import java.util.Iterator;
 3  import java.util.NoSuchElementException;
 4
 5  /**
 6     A hash set stores an unordered collection of objects, using
 7     a hash table.
 8  */
 9  public class HashSet extends AbstractSet
10  {
11     /**
12        Constructs a hash table.
13        @param bucketsLength the length of the buckets array
14     */
15     public HashSet(int bucketsLength)
16     {
17        buckets = new Node[bucketsLength];
18        size = 0;
19     }
20
21     /**
22        Tests for set membership.
23        @param x an object
24        @return true if x is an element of this set
25     */
26     public boolean contains(Object x)
27     {
28        int h = x.hashCode();
29        if (h < 0) h = -h;
30        h = h % buckets.length;
31
32        Node current = buckets[h];
33        while (current ! = null)
34        {
35           if (current.data.equals(x)) return
   true;
36           current = current.next;
37        }
```

```
38          return false;
39       }
40
41       /**
42          Adds an element to this set.
43          @param x an object
44          @return true if x is a new object, false if x was
45          already in the set
46       */
47       public boolean add(Object x)
48       {
49          int h = x.hashCode();
50          if (h < 0) h = -h;
51          h = h % buckets.length;
52
53          Node current = buckets[h];
54          while (current != null)
55          {
56             if (current.data.equals(x))
57                return false;// Already in the set
58             current = current.next;
59          }
60          Node newNode = new Node();
61          newNode.data = x;
62          newNode.next = buckets[h];
63          buckets[h] = newNode;
64          size++;
65          return true;
66       }
67
68       /**
69          Removes an object from this set.
70          @param x an object
71          @return true if x was removed from this set, false
72          if x was not an element of this set
73       */                                                    711
74       public boolean remove(Object x)                       712
75       {
76          int h = x.hashCode();
77          if (h < 0) h = -h;
78          h = h % buckets.length;
```

```
 79
 80          Node current = buckets[h];
 81          Node previous = null;
 82          while (current ! = null)
 83          {
 84             if (current.data.equals(x))
 85             {
 86                if (previous == null) buckets[h]
    = current.next;
 87                else previous.next = current.next;
 88                size--;
 89                return true;
 90             }
 91             previous = current;
 92             current = current.next;
 93          }
 94          return false;
 95       }
 96
 97       /**
 98          Returns an iterator that traverses the elements of this set.
 99          @return a hash set iterator
100       */
101       public Iterator iterator()
102       {
103          return new HashSetIterator();
104       }
105
106       /**
107          Gets the number of elements in this set.
108          @return the number of elements
109       */
110       public int size()
111       {
112          return size;
113       }
114
115       private Node[] buckets;
116       private int size;
117
118       private class Node
119       {
```

```
120         public Object data;
121         public Node next;
122      }
123
124      private class HashSetIterator implements
Iterator
125      {
126         /**
127            Constructs a hash set iterator that points to the
128            first element of the hash set.                          712
129         */                                                         713
130         public HashSetIterator()
131         {
132            current = null;
133            bucket = -1;
134            previous = null;
135            previousBucket = -1;
136         }
137
138         public boolean hasNext()
139         {
140            if (current != null && current.next !
= null)
141               return true;
142            for (int b = bucket + 1; b <
buckets.length; b++)
143               if (buckets[b] != null) return
true;
144            return false;
145         }
146
147         public Object next()
148         {
149            previous = current;
150            previousBucket = bucket;
151            if (current == null | | current.next
== null)
152            {
153               // Move to next bucket
154               bucket++;
155
156               while (bucket < buckets.length
```

```
157                      && buckets[bucket] == null)
158                   bucket++;
159                if (bucket < buckets.length)
160                   current = buckets[bucket];
161                else
162                   throw new
NoSuchElementException();
163             }
164             else // Move to next element in bucket
165                current = current.next;
166             return current.data;
167          }
168
169          public void remove()
170          {
171             if (previous != null &&
previous.next == current)
172                previous.next = current.next;
173             else if (previousBucket < bucket)
174                buckets[bucket] = current.next;
175             else
176                throw new IllegalStateException();
177             current = previous;
178             bucket = previousBucket;
179          }
180
181          private int bucket;
182          private Node current;                    713
183          private int previousBucket;              714
184          private Node previous;
185       }
186    }
```

ch16/hashtable/HashSetDemo.java

```
1  import java.util.Iterator;
2  import java.util.Set;
3
4  /**
5     This program demonstrates the hash set class.
6  */
7  public class HashSetDemo
```

```
 8  {
 9     public static void main(String[] args)
10     {
11        Set names = new HashSet(101); // 101 is a prime
12
13        names.add("Sue");
14        names.add("Harry");
15        names.add("Nina");
16        names.add("Susannah");
17        names.add("Larry");
18        names.add("Eve");
19        names.add("Sarah");
20        names.add("Adam");
21        names.add("Tony");
22        names.add("Katherine");
23        names.add("Juliet");
24        names.add("Romeo");
25        names.remove("Romeo");
26        names.remove("George");
27
28        Iterator iter = names.iterator();
29        while (iter.hasNext())
30           System.out.println(iter.next());
31     }
32  }
```

Output

```
Harry
Sue
Nina
Susannah
Larry
Eve
Sarah
Adam
Juliet
Katherine
Tony
```

SELF CHECK

5. If a hash function returns `0` for all values, will the `HashSet` work correctly?
6. What does the `hasNext` method of the `HashSetIterator` do when it has reached the end of a bucket?

16.4 Computing Hash Codes

A hash function computes an integer hash code from an object, so that different objects are likely to have different hash codes. Let us first look at how you can compute a hash code from a string. Clearly, you need to combine the character values of the string to yield some integer. You could, for example, add up the character values:

```
int h = 0;
for (int i = 0; i < s.length(); i++)
   h = h + s.charAt(i);
```

However, that would not be a good idea. It doesn't scramble the character values enough. Strings that are permutations of another (such as `"eat"` and `"tea"`) all have the same hash code.

Here is the method the standard library uses to compute the hash code for a string.

```
final int HASH_MULTIPLIER = 31;
int h = 0;
for (int i = 0; i < s.length(); i++)
   h = HASH_MULTIPLIER * h + s.charAt(i);
```

For example, the hash code of `"eat"` is

```
31 * (31 * 'e' + 'a') + 't' = 100184
```

The hash code of `"tea"` is quite different, namely

```
31 * (31 * 't' + 'e') + 'a' = 114704
```

(Use the Unicode table from Appendix B to look up the character values: `'a'` is 97, `'e'` is 101, and `'t'` is 116.)

For your own classes, you should make up a hash code that combines the hash codes of the instance fields in a similar way. For example, let us define a `hashCode` method for the `Coin` class. There are two instance fields: the coin name and the coin value. First, compute their hash code. You know how to compute the hash code of a string. To compute the hash code of a floating-point number, first wrap the floating-point number into a `Double` object, and then compute its hash code.

Define `hashCode` methods for your own classes by combining the hash codes for the instance variables.

```
class Coin
{
   public int hashCode()
   {
      int h1 = name.hashCode();
      int h2 = new Double(value).hashCode();
      . . .
   }
}
```

715

716

Then combine the two hash codes.

```
final int HASH_MULTIPLIER = 29;
int h = HASH_MULTIPLIER * h1 + h2;
return h;
```

Use a prime number as the hash multiplier—it scrambles the values better.

If you have more than two instance fields, then combine their hash codes as follows:

```
int h = HASH_MULTIPLIER * h1 + h2;
h = HASH_MULTIPLIER * h + h3;
h = HASH_MULTIPLIER * h + h4;
. . .
return h;
```

If one of the instance fields is an integer, just use the field value as its hash code.

When you add objects of your class into a hash table, you need to double-check that the `hashCode` method is *compatible* with the `equals` method of your class. Two objects that are equal must yield the same hash code:

- If `x.equals(y)`, then `x.hashCode() == y.hashCode()`

After all, if `x` and `y` are `equal` to each other, then you don't want to insert both of them into a set—sets don't store duplicates. But if their hash codes are different, `x` and `y` may end up in different buckets, and the `add` method would never notice that they are actually duplicates.

> Your `hashCode` method must be compatible with the `equals` method.

Of course, the converse of the compatibility condition is generally not true. It is possible for two objects to have the same hash code without being equal.

For the `Coin` class, the compatibility condition holds. We define two coins to be equal to each other if their names and values are equal. In that case, their hash codes will also be equal, because the hash code is computed from the hash codes of the `name` and `value` fields.

You get into trouble if your class defines an `equals` method but not a `hashCode` method. Suppose we forget to define a `hashCode` method for the `Coin` class. Then it inherits the hash code method from the `Object` superclass. That method computes a hash code from the *memory location* of the object. The effect is that any two objects are very likely to have a different hash code.

```
Coin coin1 = new Coin(0.25, "quarter");
Coin coin2 = new Coin(0.25, "quarter");
```

Now `coin1.hashCode()` is derived from the memory location of `coin1`, and `coin2.hashCode()` is derived from the memory location of `coin2`. Even though `coin1.equals(coin2)` is `true`, their hash codes differ.

However, if you define *neither* `equals` *nor* `hashCode`, then there is no problem. The `equals` method of the `Object` class considers two objects equal only if their memory location is the same. That is, the `Object` class has compatible `equals` and `hashCode` methods. Of course, then the notion of equality is very restricted: Only identical objects are considered equal. That is not necessarily a bad notion of equality: If you want to collect a set of coins in a purse, you may not want to lump coins of equal value together.

716

717

Whenever you use a hash set, you need to make sure that an appropriate hash function exists for the type of the objects that you add to the set. Check the `equals` method of your class. It tells you when two objects are considered equal. There are two possibilities. Either `equals` has been defined or it has not been defined. If `equals` has not been defined, only identical objects are considered equal. In that case, don't define `hashCode` either. However, if the `equals` method has been defined, look at its implementation. Typically, two objects are considered equal if some or all of the instance fields are equal. Sometimes, not all instance fields are used in the comparison. Two `Student` objects may be considered equal if their `studentID` fields are equal. Define the `hashCode` method to combine the hash codes of the fields that are compared in the `equals` method.

In a hash map, only the keys are hashed.

When you use a `HashMap`, only the keys are hashed. They need compatible `hashCode` and `equals` methods. The values are never hashed or compared. The reason is simple—the map only needs to find, add, and remove keys quickly.

What can you do if the objects of your class have `equals` and `hashCode` methods defined that don't work for your situation, or if you don't want to define an appropriate `hashCode` method? Maybe you can use a `TreeSet` or `TreeMap` instead. Trees are the subject of the next section.

ch16/hashcode/Coin.java

```
1  /**
2     A coin with a monetary value.
3  */
4  public class Coin
5  {
6     /**
7        Constructs a coin.
8        @param aValue the monetary value of the coin
9        @param aName the name of the coin
10    */
11    public Coin(double aValue, String aName)
12    {
```

```
13          value = aValue;
14          name = aName;
15       }
16
17       /**
18          Gets the coin value.
19          @return the value
20       */
21       public double getValue()
22       {
23          return value;
24       }
25
26       /**
27          Gets the coin name.
28          @return the name
29       */
30       public String getName()
31       {
32          return name;
33       }
34
35       public boolean equals(Object otherObject)
36       {
37          if (otherObject == null) return false;
38          if (getClass() !=
otherObject.getClass()) return false;
39          Coin other = (Coin) otherObject;
40          return value == other.value &&
name.equals(other.name);
41       }
42
43       public int hashCode()
44       {
45          int h1 = name.hashCode();
46          int h2 = new Double(value).hashCode();
47          final int HASH_MULTIPLIER = 29;
48          int h = HASH_MULTIPLIER * h1 + h2;
49          return h;
50       }
51
52       public String toString()
```

717

718

```
53      {
54         return "Coin[value = " + value +
"",name="" + name + ""]"";
55      }
56
57      private double value;
58      private String name;
59   }
```

ch16/hashcode/CoinHashCodePrinter.java

```
 1   import java.util.HashSet;
 2   import java.util.Set;
 3
 4   /**
 5      A program that prints hash codes of coins.
 6   */
 7   public class CoinHashCodePrinter
 8   {
 9      public static void main(String[] args)
10      {
11         Coin coin1 = new Coin(0.25, "quarter");
12         Coin coin2 = new Coin(0.25, "quarter");
13         Coin coin3 = new Coin(0.05, "nickel");
14
15         System.out.println("hash code of coin1="
16               + coin1.hashCode());
17         System.out.println("hash code of
coin2="
18               + coin2.hashCode());
19         System.out.println("hash code of coin3="
20               + coin3.hashCode());
21
22         Set<Coin> coins = new HashSet<Coin>();
23         coins.add(coin1);
24         coins.add(coin2);
25         coins.add(coin3);
26
27         for (Coin c : coins)
28            System.out.println(c);
29      }
30   }
```

Output

```
hash code of coin1=-1513525892
hash code of coin2=-1513525892
hash code of coin3=-1768365211
Coin[value=0.25,name=quarter]
Coin[value=0.05,name=nickel]
```

SELF CHECK

7. What is the hash code of the string `"to"`?

8. What is the hash code of `new Integer(13)`?

COMMON ERROR 16.1: Forgetting to Define hashCode

When putting elements into a hash table, make sure that the `hashCode` method is defined. (The only exception is that you don't need to define `hashCode` if `equals` isn't defined. In that case, distinct objects of your class are considered different, even if they have matching contents.)

If you forget to implement the `hashCode` method, then you inherit the `hashCode` method of the `Object` class. That method computes a hash code of the memory location of the object. For example, suppose that you do *not* define the `hashCode` method of the `Coin` class. Then the following code is likely to fail:

```
Set<Coin> coins = new HashSet<Coin>();
coins.add(new Coin(0.25, "quarter"));
// The following comparison will probably fail if hashCode not defined
if (coins.contains(new Coin(0.25, "quarter"))
   System.out.println("The set contains a
quarter.");
```

The two `Coin` objects are constructed at different memory locations, so the `hashCode` method of the `Object` class will probably compute different hash codes for them. (As always with hash codes, there is a small chance that the hash codes happen to collide.) Then the `contains` method will inspect the wrong bucket and never find the matching coin.

The remedy is to define a `hashCode` method in the `Coin` class.

16.5 Binary Search Trees

A set implementation is allowed to rearrange its elements in any way it chooses so that it can find elements quickly. Suppose a set implementation *sorts* its entries. Then it can use *binary search* to locate elements quickly. Binary search takes $O(\log(n))$ steps, where n is the size of the set. For example, binary search in an array of 1,000 elements is able to locate an element in about 10 steps by cutting the size of the search interval in half in each step.

There is just one wrinkle with this idea. We can't use an array to store the elements of a set, because insertion and removal in an array is slow; an $O(n)$ operation.

In this section we will introduce the simplest of many *treelike* data structures that computer scientists have invented to overcome that problem. Binary search trees allow fast insertion and removal of elements, and they are specially designed for fast searching.

A linked list is a one-dimensional data structure. Every node has a reference to a single successor node. You can imagine that all nodes are arranged in line. In contrast, a *tree* is made of nodes that have references to multiple nodes, called the child nodes. Because the child nodes can also have children, the data structure has a tree-like appearance. It is traditional to draw the tree upside down, like a family tree or hierarchy chart (see Figure 7). In a *binary tree*, every node has at most two children (called the *left* and *right children*); hence the name *binary*.

A binary tree consists of nodes, each of which has at most two child nodes.

Finally, a *binary search tree* is carefully constructed to have the following important property:

- The data values of *all* descendants to the left of *any* node are less than the data value stored in that node, and *all* descendants to the right have greater data values.

The tree in Figure 7 has this property. To verify the binary search property, you must check each node. Consider the node "Juliet". All descendants to the left have data before "Juliet". All descendants on the right have data after "Juliet". Move on to "Eve". There is a single descendant to the left, with data "Adam" before "Eve", and a single descendant to the right, with data "Harry" after "Eve". Check the remaining nodes in the same way.

> All nodes in a binary search tree fulfill the property that the descendants to the left have smaller data values than the node data value, and the descendants to the right have larger data values.

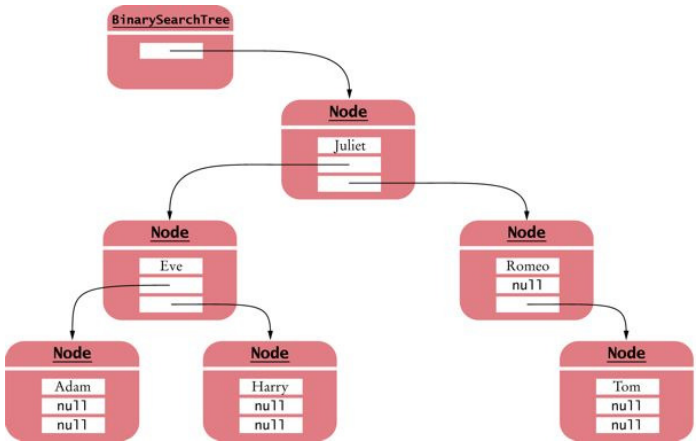
Figure 8 shows a binary tree that is not a binary search tree. Look carefully—the root node passes the test, but its two children do not.

Let us implement these tree classes. Just as you needed classes for lists and their nodes, you need one class for the tree, containing a reference to the *root node*, and a separate class for the nodes. Each node contains two references (to the left and right child nodes) and a data field. At the fringes of the tree, one or two of the child references are `null`. The `data` field has type `Comparable`, not `Object`, because you must be able to compare the values in a binary search tree in order to place them into the correct position.

720

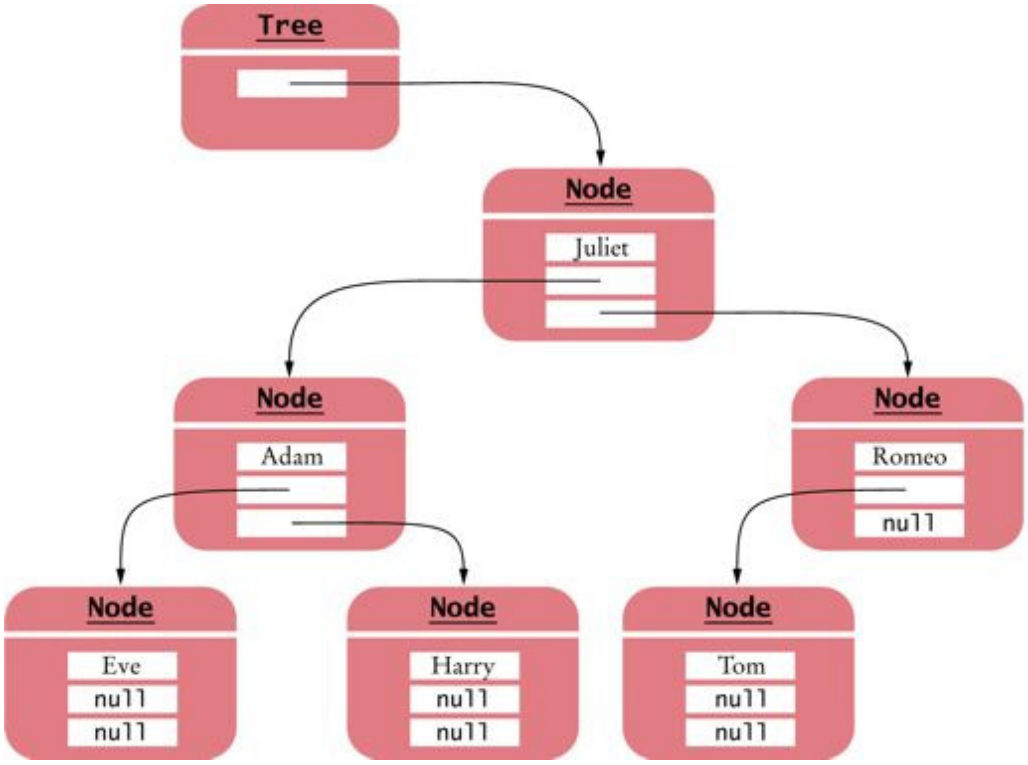
721

Figure 7



A Binary Search Tree

Figure 8



A Binary Tree That Is Not a Binary Search Tree

721

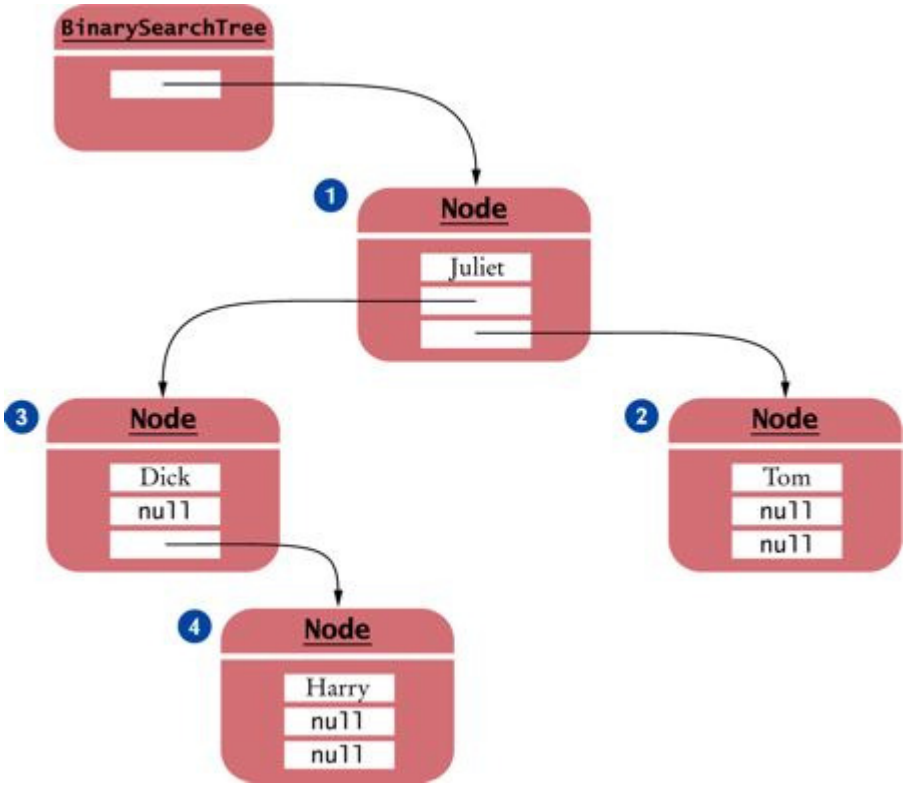
722

```
public class BinarySearchTree
{
   public BinarySearchTree() { . . . }
   public void add(Comparable obj) { . . . }
   . . .
   private Node root;
   private class Node
   {
      public void addNode(Node newNode){ . . . }
      . . .
      public Comparable data;
      public Node left;
      public Node right;
   }
}
```

To insert data into the tree, use the following algorithm:

- If you encounter a non-`null` node reference, look at its `data` value. If the `data` value of that node is larger than the one you want to insert, continue the process with the left child. If the existing `data` value is smaller, continue the process with the right child.
- If you encounter a `null` node reference, replace it with the new node.

Figure 9



Binary Search Tree After Four Insertions

722

723

For example, consider the tree in Figure 9. It is the result of the following statements:

```
BinarySearchTree tree = new BinarySearchTree();
tree.add("Juliet");
```

```
tree.add("Tom");
tree.add("Dick");
tree.add("Harry");
```

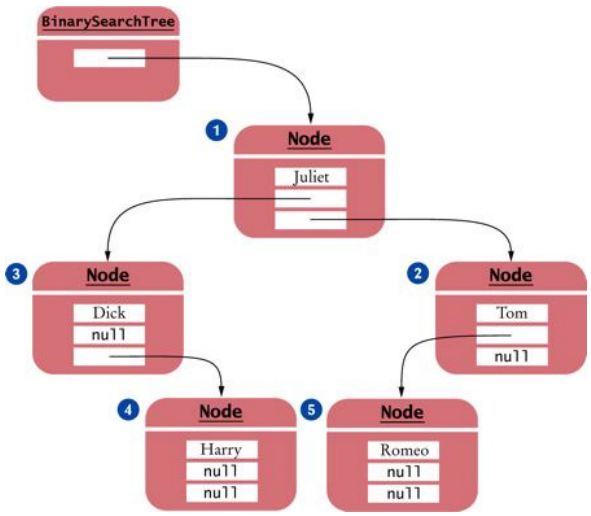
We want to insert a new element `Romeo` into it.

```
tree.add("Romeo");
```

Start with the root, `Juliet`. `Romeo` comes after `Juliet`, so you move to the right subtree. You encounter the node `Tom`. `Romeo` comes before `Tom`, so you move to the left subtree. But there is no left subtree. Hence, you insert a new `Romeo` node as the left child of `Tom` (see Figure 10).

You should convince yourself that the resulting tree is still a binary search tree. When `Romeo` is inserted, it must end up as a right descendant of `Juliet`—that is what the binary search tree condition means for the root node `Juliet`. The root node doesn't care where in the right subtree the new node ends up. Moving along to `Tom`, the right child of `Juliet`, all it cares about is that the new node `Romeo` ends up somewhere on its left. There is nothing to its left, so `Romeo` becomes the new left child, and the resulting tree is again a binary search tree.

Figure 10



Binary Search Tree After Five Insertions

Here is the code for the `add` method of the `BinarySearchTree` class:

```
public class BinarySearchTree
{
   . . .
   public void add(Comparable obj)
   {
      Node newNode = new Node();
      newNode.data = obj;
      newNode.left = null;
      newNode.right = null;
      if (root == null) root = newNode;
      else root.addNode(newNode);
   }
   . . .
}
```

If the tree is empty, simply set its root to the new node. Otherwise, you know that the new node must be inserted somewhere within the nodes, and you can ask the root node to perform the insertion. That node object calls the `addNode` method of the `Node` class, which checks whether the new object is less than the object stored in the node. If so, the element is inserted in the left subtree; if not, it is inserted in the right subtree:

```
private class Node
{
   . . .
   public void addNode(Node newNode)
   {
      int comp = newNode.data.compareTo(data);
      if (comp < 0)
      {
         if (left == null) left = newNode;
         else left.addNode(newNode);
      }
      else if (comp > 0)
      {
         if (right == null) right = newNode;
         else right.addNode(newNode);
      }
   }
   . . .
}
```

Let us trace the calls to `addNode` when inserting `Romeo` into the tree in Figure 9. The first call to `addNode` is

```
root.addNode(newNode)
```

Because `root` points to `Juliet`, you compare `Juliet` with `Romeo` and find that you must call

```
root.right.addNode(newNode)
```

The node `root.right` is `Tom`. Compare the data values again (`Tom` vs. `Romeo`) and find that you must now move to the left. Since `root.right.left` is `null`, set `root.right.left` to `newNode`, and the insertion is complete (see Figure 10).

724

725

Unlike a linked list or an array, and like a hash table, a binary tree has no *insert positions*. You cannot select the position where you would like to insert an element into a binary search tree. The data structure is *self-organizing;* that is, each element finds its own place.

We will now discuss the removal algorithm. Our task is to remove a node from the tree. Of course, we must first *find* the node to be removed. That is a simple matter, due to the characteristic property of a binary search tree. Compare the data value to be removed with the data value that is stored in the root node. If it is smaller, keep looking in the left subtree. Otherwise, keep looking in the right subtree.

Let us now assume that we have located the node that needs to be removed. First, let us consider an easy case, when that node has only one child (see Figure 11).

To remove the node, simply modify the parent link that points to the node so that it points to the child instead.

If the node to be removed has no children at all, then the parent link is simply set to `null`.

> When removing a node with only one child from a binary search tree, the child replaces the node to be removed.

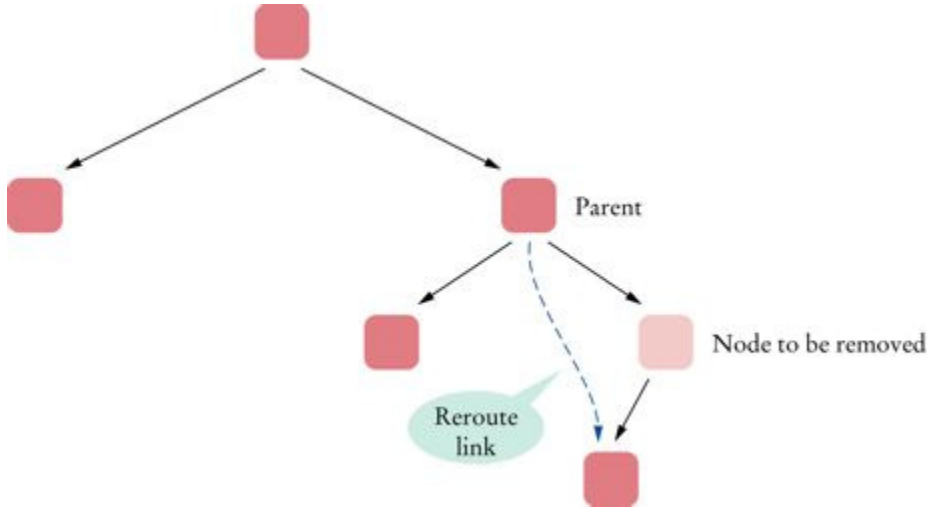
The case in which the node to be removed has two children is more challenging. Rather than removing the node, it is easier to replace its data value with the next

larger value in the tree. That replacement preserves the binary search tree property. (Alternatively, you could use the largest element of the left subtree—see Exercise P16.16).

> When removing a node with two children from a binary search tree, replace it with the smallest node of the right subtree.

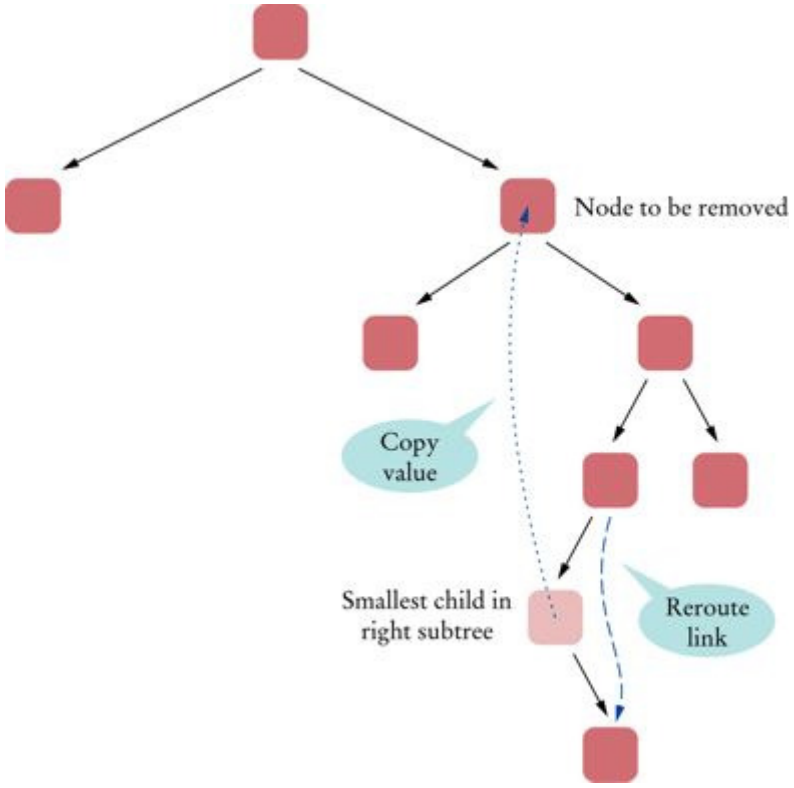
To locate the next larger value, go to the right subtree and find its smallest data value. Keep following the left child links. Once you reach a node that has no left child, you have found the node containing the smallest data value of the subtree. Now remove that node—it is easily removed because it has at most one child to the right. Then store its data value in the original node that was slated for removal. Figure 12 shows the details. You will find the complete code at the end of this section.

Figure 11



Removing a Node with One Child

Figure 12



Removing a Node with Two Children

At the end of this section, you will find the source code for the `BinarySearchTree` class. It contains the `add` and `remove` methods that we just described, as well as a `find` method that tests whether a value is present in a binary search tree, and a `print` method that we will analyze in the following section.

Now that you have seen the implementation of this complex data structure, you may well wonder whether it is any good. Like nodes in a list, nodes are allocated one at a time. No existing elements need to be moved when a new element is inserted in the tree; that is an advantage. How fast insertion is, however, depends on the shape of the tree. If the tree is *balanced*—that is, if each node has approximately as many descendants on the left as on the right—then insertion is very fast, because about half of the nodes are eliminated in each step. On the other hand, if the tree happens to be

unbalanced, then insertion can be slow—perhaps as slow as insertion into a linked list. (See Figure 13.)

If a binary search tree is balanced, then adding an element takes $O(\log(n))$ time.

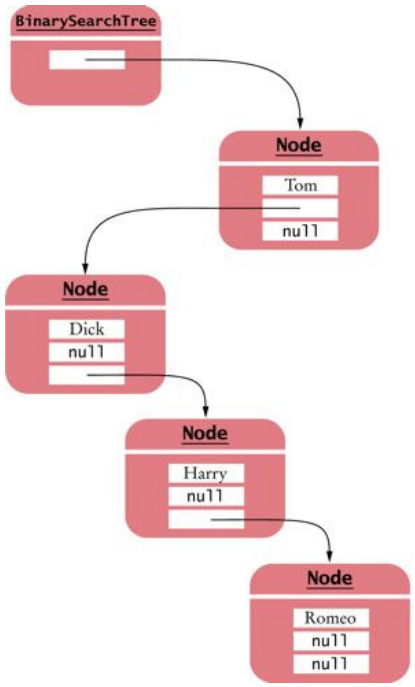
If new elements are fairly random, the resulting tree is likely to be well balanced. However, if the incoming elements happen to be in sorted order already, then the resulting tree is completely unbalanced. Each new element is inserted at the end, and the entire tree must be traversed every time to find that end!

Binary search trees work well for random data, but if you suspect that the data in your application might be sorted or have long runs of sorted data, you should not use a binary search tree. There are more sophisticated tree structures whose methods keep trees balanced at all times. In these tree structures, one can guarantee that finding, adding, and removing elements takes $O(\log(n))$ time. To learn more about those advanced data structures, you may want to enroll in a course about data structures.

726

727

Figure 13



An Unbalanced Binary Search Tree

The standard Java library uses *red-black trees*, a special form of balanced binary trees, to implement sets and maps. You will see in Section 16.7 what you need to do to use the `TreeSet` and `TreeMap` classes. For information on how to implement a red-black tree yourself, see [1].

ch16/tree/BinarySearchTree.java

```
1  /**
2     This class implements a binary search tree whose
3     nodes hold objects that implement the Comparable
4     interface.
5  */
6  public class BinarySearchTree
7  {                                                          727
8  /**                                                        728
9     Constructs an empty tree.
10 */
11 public BinarySearchTree()
12 {
13    root = null;
14 }
15
16 /**
17       Inserts a new node into the tree.
18       @param obj the object to insert
19 */
20   public void add(Comparable obj)
21   {
22      Node newNode = new Node();
23      newNode.data = obj;
24      newNode.left = null;
25      newNode.right = null;
26      if (root == null) root = newNode;
27      else root.addNode(newNode);
28   }
29
30 /**
31       Tries to find an object in the tree.
32       @param obj the object to find
```

```
33          @return true if the object is contained in the tree
34       */
35       public boolean find(Comparable obj)
36       {
37          Node current = root;
38          while (current != null)
39          {
40             int d = current.data.compareTo(obj);
41             if (d == 0) return true;
42             else if (d > 0) current =
current.left;
43             else current = current.right;
44          }
45          return false;
46       }
47
48       /**
49          Tries to remove an object from the tree. Does nothing
50          if the object is not contained in the tree.
51          @param obj the object to remove
52       */
53       public void remove(Comparable obj)
54       {
55          // Find node to be removed
56
57          Node toBeRemoved = root;
58          Node parent = null;
59          boolean found = false;
60          while (!found && toBeRemoved != null)
61          {                                                    728
62             int d =                                           729
toBeRemoved.data.compareTo(obj);
63             if (d == 0) found = true;
64             else
65             {
66                parent = toBeRemoved;
67                if (d > 0) toBeRemoved =
toBeRemoved.left;
68                else toBeRemoved =
toBeRemoved.right;
69             }
70          }
```

```
 71
 72          if (!found) return;
 73
 74          // toBeRemoved contains obj
 75
 76          // If one of the children is empty, use the other
 77
 78          if (toBeRemoved.left == null ||
    toBeRemoved.right == null)
 79          {
 80             Node newChild;
 81             if (toBeRemoved.left == null)
 82                newChild = toBeRemoved.right;
 83             else
 84                newChild = toBeRemoved.left;
 85
 86             if (parent == null) // Found in root
 87                root = newChild;
 88             else if (parent.left == toBeRemoved)
 89                parent.left = newChild;
 90             else
 91                parent.right = newChild;
 92             return;
 93          }
 94
 95          // Neither subtree is empty
 96
 97          // Find smallest element of the right subtree
 98
 99          Node smallestParent = toBeRemoved;
100          Node smallest = toBeRemoved.right;
101          while (smallest.left != null)
102          {
103             smallestParent = smallest;
104             smallest = smallest.left;
105          }
106
107          // smallest contains smallest child in right subtree
108
109          // Move contents, unlink child
110
```

```
111         toBeRemoved.data = smallest.data;
112         smallestParent.left = smallest.right;
113      }
114                                                                    729
115      /**                                                           730
116         Prints the contents of the tree in sorted order.
117      */
118      public void print()
119      {
120         if (root != null)
121            root.printNodes();
122         System.out.println();
123      }
124
125      private Node root;
126
127      /**
128         A node of a tree stores a data item and references
129         to the child nodes to the left and to the right.
130      */
131      private class Node
132      {
133         /**
134            Inserts a new node as a descendant of this node.
135            @param newNode the node to insert
136         */
137         public void addNode(Node newNode)
138         {
139            int comp =
newNode.data.compareTo(data);
140            if (comp < 0)
141            {
142               if (left == null) left = newNode;
143               else left.addNode(newNode);
144            }
145            if (comp > 0)
146            {
147               if (right == null) right = newNode;
148               else right.addNode(newNode);
149            }
150         }
151
```

```
152         /**
153            Prints this node and all of its descendants
154            in sorted order.
155         */
156         public void printNodes()
157         {
158            if (left != null)
159               left.printNodes();
160            System.out.println(data + " ");
161            if (right != null)
162               right.printNodes();
163         }
164
165         public Comparable data;
166         public Node left;
167         public Node right;
168      }
169   }
```

730

731

SELF CHECK

9. What is the difference between a tree, a binary tree, and a balanced binary tree?

10. Give an example of a string that, when inserted into the tree of Figure 10, becomes a right child of `Romeo`.

16.6 Tree Traversal

Now that the data are inserted in the tree, what can you do with them? It turns out to be surprisingly simple to print all elements in sorted order. You *know* that all data in the left subtree of any node must come before the node and before all data in the right subtree. That is, the following algorithm will print the elements in sorted order:

1. Print the left subtree.
2. Print the data.
3. Print the right subtree.

Let's try this out with the tree in Figure 10. The algorithm tells us to

1. Print the left subtree of `Juliet`; that is, `Dick` and descendants.
2. Print `Juliet`.
3. Print the right subtree of `Juliet`; that is, `Tom` and descendants.

How do you print the subtree starting at `Dick`?

1. Print the left subtree of `Dick`. There is nothing to print.
2. Print `Dick`.
3. Print the right subtree of `Dick`, that is, `Harry`.

That is, the left subtree of `Juliet` is printed as

```
Dick Harry
```

The right subtree of `Juliet` is the subtree starting at `Tom`. How is it printed? Again, using the same algorithm:

1. Print the left subtree of `Tom`, that is, `Romeo`.
2. Print `Tom`.
3. Print the right subtree of `Tom`. There is nothing to print.

Thus, the right subtree of `Juliet` is printed as

```
Romeo Tom
```

731

732

Now put it all together: the left subtree, `Juliet`, and the right subtree:

```
Dick Harry Juliet Romeo Tom
```

The tree is printed in sorted order.

Let us implement the `print` method. You need a worker method `printNodes` of the `Node` class:

```
private class Node
{
   . . .
   public void printNodes()
```

```
      {
         if (left != null)
            left.printNodes();
         System.out.print(data + " ");
         if (right != null)
            right.printNodes();
      }
      . . .
   }
```

To print the entire tree, start this recursive printing process at the root, with the following method of the `BinarySearchTree` class.

```
public class BinarySearchTree
{
   . . .
   public void print()
   {
      if (root != null)
         root.printNodes();
      System.out.println();
   }
   . . .
}
```

This visitation scheme is called *inorder traversal*. There are two other traversal schemes, called *preorder traversal* and *postorder traversal*.

Tree traversal schemes include preorder traversal, inorder traversal, and postorder traversal.

In preorder traversal,

- Visit the root
- Visit the left subtree
- Visit the right subtree

In postorder traversal,

- Visit the left subtree

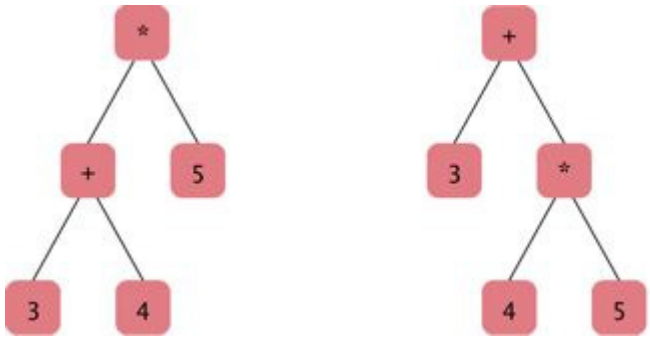
- Visit the right subtree
- Visit the root

These two visitation schemes will not print the tree in sorted order. However, they are important in other applications of binary trees. Here is an example.

732

733

Figure 14



Expression Trees

In Chapter 13, we presented an algorithm for parsing arithmetic expressions such as

```
(3 + 4) * 5
3 + 4 * 5
```

It is customary to draw these expressions in tree form—see Figure 14. If all operators have two arguments, then the resulting tree is a binary tree. Its leaves store numbers, and its interior nodes store operators.

Note that the expression trees describe the order in which the operators are applied.

This order becomes visible when applying the postorder traversal of the expression tree. The first tree yields

```
3 4 + 5 *
```

whereas the second tree yields

```
3 4 5 * +
```

You can interpret these sequences as instructions for a stack-based calculator. A number means:

- Push the number on the stack.

An operator means:

- Pop the top two numbers off the stack.
- Apply the operator to these two numbers.
- Push the result back on the stack.

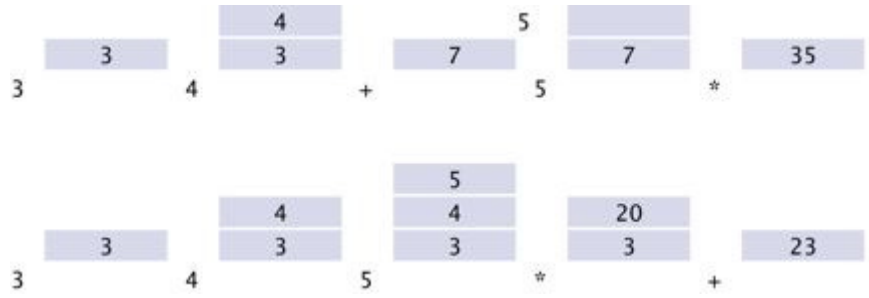
Figure 15 shows the computation sequences for the two expressions.

> Postorder traversal of an expression tree yields the instructions for evaluating the expression on a stack-based calculator.

This observation yields an algorithm for evaluating arithmetic expressions. First, turn the expression into a tree. Then carry out a postorder traversal of the expression tree and apply the operations in the given order. The result is the value of the expression.

733
734

Figure 15



A Stack-Based Calculator

SELF CHECK

11. What are the inorder traversals of the two trees in Figure 14?

12. Are the trees in Figure 14 binary search trees?

RANDOM FACT 16.1: Reverse Polish Notation

In the 1920s, the Polish mathematician Jan Łukasiewicz realized that it is possible to dispense with parentheses in arithmetic expressions, provided that you write the operators *before* their arguments. For example,

Standard Notation	Łukasiewicz Notation
`3 + 4`	`+ 3 4`
`3 + 4 * 5`	`+ 3 * 4 5`
`3 * (4 + 5)`	`* 3 + 4 5`
`(3 + 4) * 5`	`* + 3 4 5`
`3 + 4 + 5`	`+ + 3 4 5`

The Łukasiewicz notation might look strange to you, but that is just an accident of history. Had earlier mathematicians realized its advantages, schoolchildren today would not learn an inferior notation with arbitrary precedence rules and parentheses.

Of course, an entrenched notation is not easily displaced, even when it has distinct disadvantages, and Łukasiewicz's discovery did not cause much of a stir for about 50 years.

However, in 1972, Hewlett-Packard introduced the HP 35 calculator that used *reverse Polish notation* or RPN. RPN is simply Łukasiewicz's notation in reverse, with the operators after their arguments. For example, to compute `3 + 4 * 5`, you enter `3 4 5 * +`. RPN calculators have no keys labeled with parentheses or an equals symbol. There is just a key labeled ENTER to push a number onto a stack. For that reason, Hewlett-Packard's marketing department used to refer to their product as “the calculators that have no equal”. Indeed, the Hewlett-Packard calculators were a great advance over competing models that were unable to handle algebraic notation, leaving users with no other choice but to write intermediate results on paper.

734
735

Over time, developers of high-quality calculators have adapted to the standard algebraic notation rather than forcing its users to learn a new notation. However, those users who have made the effort to learn RPN tend to be fanatic proponents, and to this day, some Hewlett-Packard calculator models still support it.

16.7 Using Tree Sets and Tree Maps

Both the `HashSet` and the `TreeSet` classes implement the `Set` interface. Thus, if you need a set of objects, you have a choice.

If you have a good hash function for your objects, then hashing is usually faster than tree-based algorithms. But the balanced trees used in the `TreeSet` class can *guarantee* reasonable performance, whereas the `HashSet` is entirely at the mercy of the hash function.

The `TreeSet` class uses a form of balanced binary tree that guarantees that adding and removing an element takes $O(\log(n))$ time.

If you don't want to define a hash function, then a tree set is an attractive option. Tree sets have another advantage: The iterators visit elements in *sorted order* rather than the completely random order given by the hash codes.

To use a `TreeSet`, your objects must belong to a class that implements the `Comparable` interface or you must supply a `Comparator` object. That is the same

requirement that you saw in Section 14.8 for using the `sort` and `binarySearch` methods in the standard library.

To use a tree set, the elements must be comparable.

To use a `TreeMap`, the same requirement holds for the *keys*. There is no requirement for the values.

For example, the `String` class implements the `Comparable` interface. The `compareTo` method compares strings in dictionary order. Thus, you can form tree sets of strings, and use strings as keys for tree maps.

735

736

If the class of the tree set elements doesn't implement the `Comparable` interface, or the sort order of the `compareTo` method isn't the one you want, then you can define your own comparison by supplying a `Comparator` object to the `TreeSet` or `TreeMap` constructor. For example,

```
Comparator comp = new CoinComparator();
Set s = new TreeSet(comp);
```

As described in Advanced Topic 14.5, a `Comparator` object compares two elements and returns a negative integer if the first is less than the second, zero if they are identical, and a positive value otherwise. The example program at the end of this section constructs a `TreeSet` of `Coin` objects, using the coin comparator of Advanced Topic 14.5.

ch16/treeset/TreeSetTester.java

```
1  import java.util.Comparator;
2  import java.util.Set;
3  import java.util.TreeSet;
4
5  /**
6     A program to a test a tree set with a comparator for coins.
7  */
8  public class TreeSetTester
9  {
10    public static void main(String[] args)
11    {
12       Coin coin1 = new Coin(0.25, "quarter");
```

```
13          Coin coin2 = new Coin(0.25, "quarter");
14          Coin coin3 = new Coin(0.01, "penny");
15          Coin coin4 = new Coin(0.05, "nickel");
16
17          class CoinComparator implements
Comparator<Coin>
18          {
19             public int compare(Coin first, Coin
second)
20             {
21                if (first.getValue() <
second.getValue()) return -1;
22                if (first.getValue() ==
second.getValue()) return 0;
23                return 1;
24             }
25          }
26
27          Comparator<Coin> comp = new
CoinComparator();
28          Set<Coin> coins = new
TreeSet<Coin>(comp);
29          coins.add(coin1);
30          coins.add(coin2);
31          coins.add(coin3);
32          coins.add(coin4);
33
34          for (Coin c : coins)
35             System.out.print(c.getValue() + " ");
36          System.out.println("Expected: 0.01 0.05
0.25");
37       }
38    }
```

736

737

Output

```
0.01 0.05 0.25
Expected: 0.01 0.05 0.25
```

SELF CHECK

13. When would you choose a tree set over a hash set?

14. Suppose we define a coin comparator whose `compare` method always returns 0. Would the `TreeSet` function correctly?

How To 16.1: Choosing a Container

Suppose you need to store objects in a container. You have now seen a number of different data structures. This How To reviews how to pick an appropriate container for your application.

Step 1 Determine how you access the elements.

You store elements in a container so that you can later retrieve them. How do you want to access individual elements? You have several choices.

- It doesn't matter. Elements are always accessed "in bulk", by visiting all elements and doing something with them.
- Access by key. Elements are accessed by a special key. *Example*: Retrieve a bank account by the account number.
- Access by integer index. Elements have a position that is naturally an integer or a pair of integers. *Example*: A piece on a chess board is accessed by a row and column index.

If you need keyed access, use a map. If you need access by integer index, use an array list or array. For an index pair, use a two-dimensional array.

Step 2 Determine whether element order matters.

When you retrieve elements from a container, do you care about the order in which they are retrieved? You have several choices.

- It doesn't matter. As long as you get to visit all elements, you don't care in which order.
- Elements must be sorted.
- Elements must be in the same order in which they were inserted.

To keep elements sorted, use a `TreeSet`. To keep elements in the order in which you inserted them, use a `LinkedList, ArrayList`, or array.

Step 3 Determine which operations must be fast.

You have several choices.

- It doesn't matter. You collect so few elements that you aren't concerned about speed.
- Adding and removing elements must be fast.
- Finding elements must be fast.

737

738

Linked lists allow you to add and remove elements efficiently, provided you are already near the location of the change. Changing either end of the linked list is always fast.

If you need to find an element quickly, use a set.

At this point, you should have narrowed down your selection to a particular container. If you answered “It doesn't matter” for each of the choices, then just use an `ArrayList`. It's a simple container that you already know well.

Step 4 For sets and maps, choose between hash tables and trees.

If you decided that you need a set or map, you need to pick a particular implementation, either a hash table or a tree.

If your elements (or keys, in case of a map) are strings, use a hash table. It's more efficient.

If your elements or keys belong to a type that someone else defined, check whether the class implements its own `hashCode` and `equals` methods. The inherited `hashCode` method of the `Object` class takes only the object's memory address into account, not its contents. If there is no satisfactory `hashCode` method, then you must use a tree.

If your elements or keys belong to your own class, you usually want to use hashing. Define a `hashCode` and compatible `equals` method.

Step 5 If you use a tree, decide whether to supply a comparator.

Look at the class of the elements or keys that the tree manages. Does that class implement the `Comparable` interface? If so, is the sort order given by the `compareTo` method the one you want? If yes, then you don't need to do anything further. If no, then you must define a class that implements the `Comparator` interface and define the `compare` method. Supply an object of the comparator class to the `TreeSet` or `TreeMap` constructor.

RANDOM FACT 16.2: Software Piracy

As you read this, you have written a few computer programs, and you have experienced firsthand how much effort it takes to write even the humblest of programs. Writing a real software product, such as a financial application or a computer game, takes a lot of time and money. Few people, and fewer companies, are going to spend that kind of time and money if they don't have a reasonable chance to make more money from their effort. (Actually, some companies give away their software in the hope that users will upgrade to more elaborate paid versions. Other companies give away the software that enables users to read and use files but sell the software needed to create those files. Finally, there are individuals who donate their time, out of enthusiasm, and produce programs that you can copy freely.)

When selling software, a company must rely on the honesty of its customers. It is an easy matter for an unscrupulous person to make copies of computer programs without paying for them. In most countries that is illegal. Most governments provide legal protection, such as copyright laws and patents, to encourage the development of new products. Countries that tolerate widespread piracy have found that they have an ample cheap supply of foreign software, but no local manufacturers willing to design good software for their own citizens, such as word processors in the local script or financial programs adapted to the local tax laws.

When a mass market for software first appeared, vendors were enraged by the money they lost through piracy. They tried to fight back by various schemes to ensure that only the legitimate owner could use the software. Some manufacturers used *key disks*: disks with special patterns of holes burned in by a laser, which

couldn't be copied. Others used *dongles*: devices that are attached to a printer port. Legitimate users hated these measures. They paid for the software, but they had to suffer through the inconvenience of inserting a key disk every time they started the software or having multiple dongles stick out from their computer. In the United States, market pressures forced most vendors to give up on these copy protection schemes, but they are still commonplace in other parts of the world.

Because it is so easy and inexpensive to pirate software, and the chance of being found out is minimal, you have to make a moral choice for yourself. If a package that you would really like to have is too expensive for your budget, do you steal it, or do you stay honest and get by with a more affordable product?

Of course, piracy is not limited to software. The same issues arise for other digital products as well. You may have had the opportunity to obtain copies of songs or movies without payment. Or you may have been frustrated by a copy protection device on your music player that made it difficult for you to listen to songs that you paid for. Admittedly, it can be difficult to have a lot of sympathy for a musical ensemble whose publisher charges a lot of money for what seems to have been very little effort on their part, at least when compared to the effort that goes into designing and implementing a software package. Nevertheless, it seems only fair that artists and authors receive some compensation for their efforts. How to pay artists, authors, and programmers fairly, without burdening honest customers, is an unsolved problem at the time of this writing, and many computer scientists are engaged in research in this area.

16.8 Priority Queues

In Section 15.4, you encountered two common abstract data types: stacks and queues. Another important abstract data type, the *priority queue*, collects elements, each of which has a *priority*. A typical example of a priority queue is a collection of work requests, some of which may be more urgent than others.

Unlike a regular queue, the priority queue does not maintain a first-in, first-out discipline. Instead, elements are retrieved according to their priority. In other words, new items can be inserted in any order. But whenever an item is removed, that item has highest priority.

When removing an element from a priority queue, the element with the highest priority is retrieved.

It is customary to give low values to high priorities, with priority 1 denoting the highest priority. The priority queue extracts the *minimum* element from the queue.

For example, consider this sample code:

```
PriorityQueue<WorkOrder> q = new
PriorityQueue<WorkOrder>;
q.add(new WorkOrder(3, "Shampoo carpets"));
q.add(new WorkOrder(1, "Fix overflowing sink"));
q.add(new WorkOrder(2, "Order cleaning supplies"));
```

When calling `q.remove()` for the first time, the work order with priority 1 is removed. The next call to `q.remove()` removes the work order whose priority is highest among those remaining in the queue—in our example, the work order with priority 2.

The standard Java library supplies a `PriorityQueue` class that is ready for you to use. Later in this chapter, you will learn how to supply your own implementation.

739

740

Keep in mind that the priority queue is an *abstract* data type. You do not know how a priority queue organizes its elements. There are several concrete data structures that can be used to implement priority queues.

Of course, one implementation comes to mind immediately. Just store the elements in a linked list, adding new elements to the head of the list. The `remove` method then traverses the linked list and removes the element with the highest priority. In this implementation, adding elements is quick, but removing them is slow.

Another implementation strategy is to keep the elements in sorted order, for example in a binary search tree. Then it is an easy matter to locate and remove the largest element. However, another data structure, called a heap, is even more suitable for implementing priority queues.

16.9 Heaps

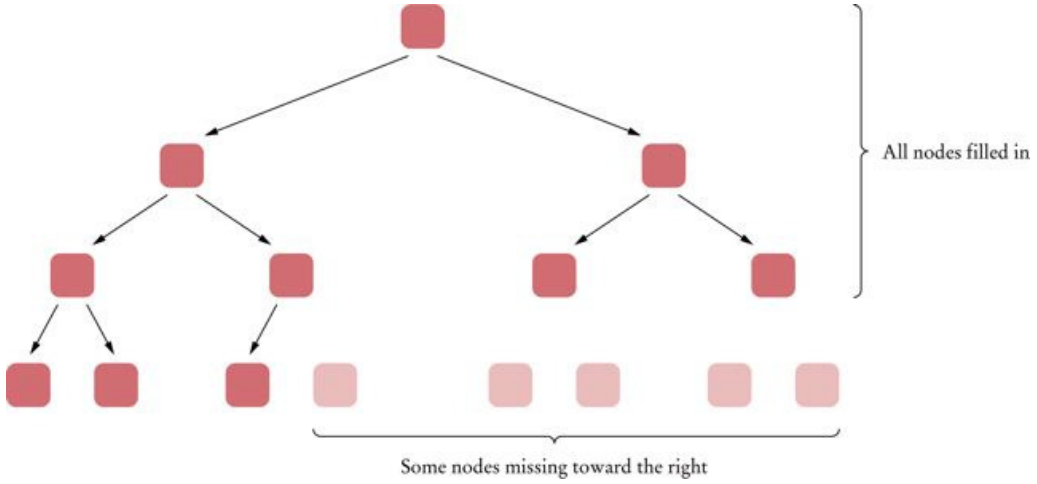
A *heap* (or, for greater clarity, *min-heap*) is a binary tree with two special properties.

1. A heap is *almost complete*: all nodes are filled in, except the last level may have some nodes missing toward the right (see Figure 16).
2. The tree fulfills the *heap property*: all nodes store values that are at most as large as the values stored in their descendants (see Figure 17).

It is easy to see that the heap property ensures that the smallest element is stored in the root.

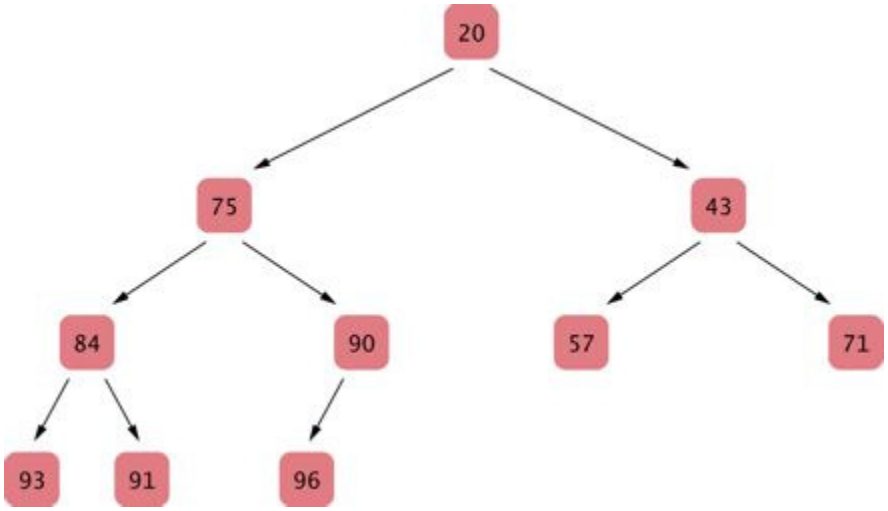
> A heap is an almost complete tree in which the values of all nodes are at most as large as those of their descendants.

Figure 16



An Almost Complete Tree

Figure 17



A Heap

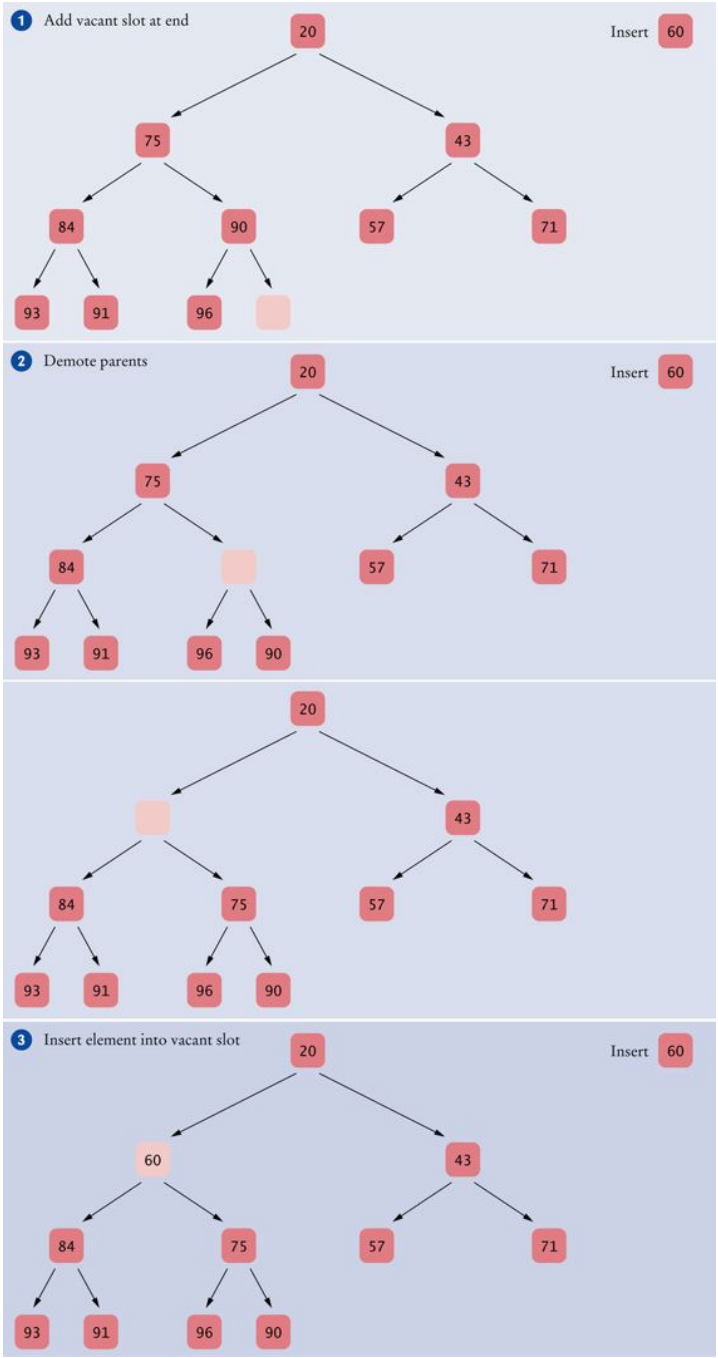
A heap is superficially similar to a binary search tree, but there are two important differences.

1. The shape of a heap is very regular. Binary search trees can have arbitrary shapes.
2. In a heap, the left and right subtrees both store elements that are larger than the root element. In contrast, in a binary search tree, smaller elements are stored in the left subtree and larger elements are stored in the right subtree.

Suppose we have a heap and want to insert a new element. Afterwards, the heap property should again be fulfilled. The following algorithm carries out the insertion (see Figure 18).

1. First, add a vacant slot to the end of the tree.

Figure 18



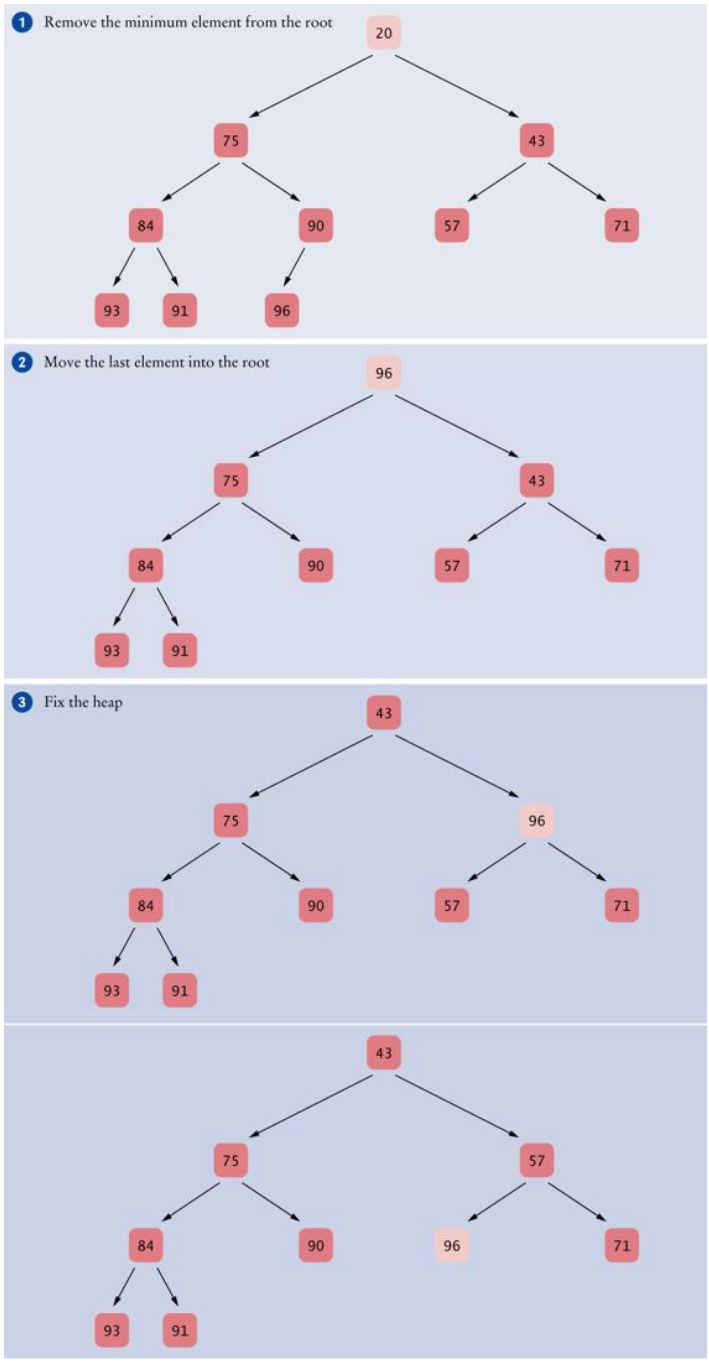
Inserting an Element into a Heap

2. Next, demote the parent of the empty slot if it is larger than the element to be inserted. That is, move the parent value into the vacant slot, and move the vacant slot up. Repeat this demotion as long as the parent of the vacant slot is larger than the element to be inserted. (See Figure 18 continued.)
3. At this point, either the vacant slot is at the root, or the parent of the vacant slot is smaller than the element to be inserted. Insert the element into the vacant slot.

We will not consider an algorithm for removing an arbitrary node from a heap. The only node that we will remove is the root node, which contains the minimum of all of the values in the heap. Figure 19 shows the algorithm in action.

1. Extract the root node value.

Figure 19



Removing the Minimum Value from a Heap

744

2. Move the value of the last node of the heap into the root node, and remove the last node. Now the heap property may be violated for the root node, because one or both of its children may be smaller.

3. Promote the smaller child of the root node. (See Figure 19 continued.) Now the root node again fulfills the heap property. Repeat this process with the demoted child. That is, promote the smaller of its children. Continue until the demoted child has no smaller children. The heap property is now fulfilled again. This process is called "fixing the heap".

Inserting and removing heap elements is very efficient. The reason lies in the balanced shape of a heap. The insertion and removal operations visit at most h nodes, 744 745
where h is the height of the tree. A heap of height h contains at least 2^{h-1} elements, but less than 2^h elements. In other words, if n is the number of elements, then

$$2^{h-1} \leq n < 2^h$$

or

$$h - 1 \leq \log_2(n) < h$$

This argument shows that the insertion and removal operations in a heap with n elements take $O(\log(n))$ steps.

> Inserting or removing a heap element is an $O(\log(n))$ operation.

Contrast this finding with the situation of binary search trees. When a binary search tree is unbalanced, it can degenerate into a linked list, so that in the worst case insertion and removal are $O(n)$ operations.

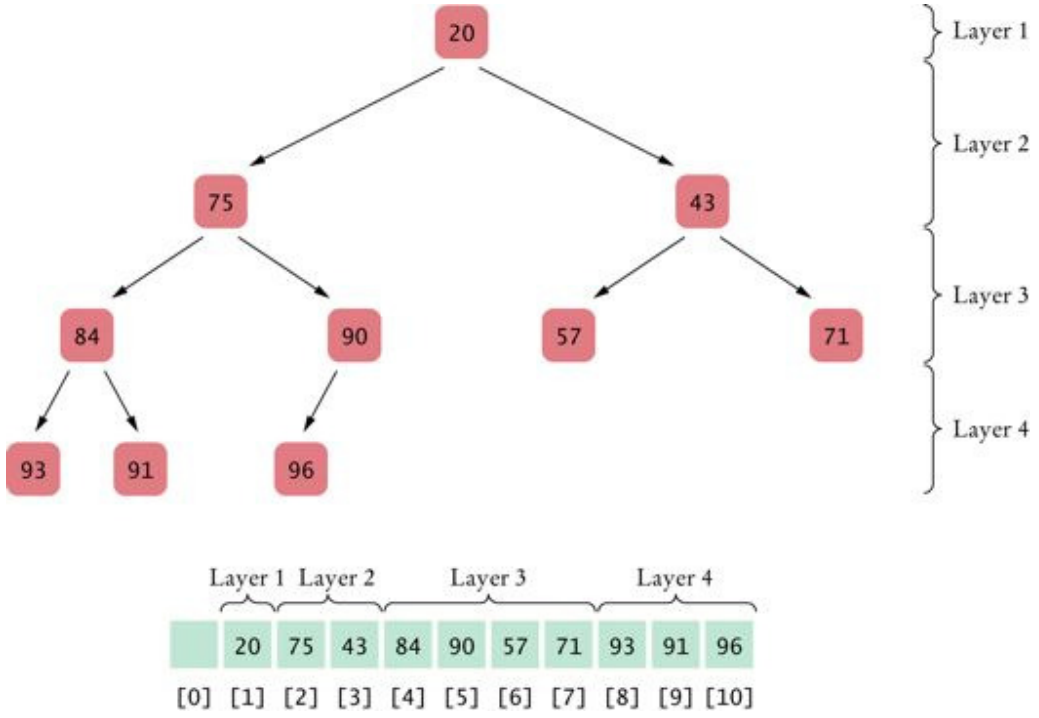
> The regular layout of a heap makes it possible to store heap nodes efficiently in an array.

Heaps have another major advantage. Because of the regular layout of the heap nodes, it is easy to store the node values in an array. First store the first layer, then the second, and so on (see Figure 20). For convenience, we leave the 0 element of the

array empty. Then the child nodes of the node with index i have index $2 \cdot i$ and $2 \cdot i + 1$, and the parent node of the node with index i has index $i/2$. For example, as you can see in Figure 20, the children of node 4 are nodes 8 and 9, and the parent is node 2.

Storing the heap values in an array may not be intuitive, but it is very efficient. There is no need to allocate individual nodes or to store the links to the child nodes. Instead, child and parent positions can be determined by very simple computations.

Figure 20



Storing a Heap in an Array

745

746

The program at the end of this section contains an implementation of a heap. For greater clarity, the computation of the parent and child index positions is carried out in methods `getParentIndex`, `getLeftChildIndex`, and `getRightChildIndex`. For greater efficiency, the method calls could be avoided by using expressions `index / 2`, `2 * index`, and `2 * index + 1` directly.

In this section, we have organized our heaps such that the smallest element is stored in the root. It is also possible to store the largest element in the root, simply by

reversing all comparisons in the heap-building algorithm. If there is a possibility of misunderstanding, it is best to refer to the data structures as min-heap or max-heap.

The test program demonstrates how to use a min-heap as a priority queue.

ch16/pqueue/MinHeap.java

```
 1  import java.util.*;
 2
 3  /**
 4     This class implements a heap.
 5  */
 6  public class MinHeap
 7  {
 8     /**
 9        Constructs an empty heap.
10     */
11     public MinHeap()
12     {
13        elements = new ArrayList<Comparable>();
14        elements.add(null);
15     }
16
17     /**
18        Adds a new element to this heap.
19        @param newElement the element to add
20     */
21     public void add(Comparable newElement)
22     {
23        // Add a new leaf
24        elements.add(null);
25        int index = elements.size() - 1;
26
27        // Demote parents that are larger than the new element
28        while (index > 1
29              &&
   getParent(index).compareTo(newElement) > 0)
30        {
31           elements.set(index,
   getParent(index));
32           index = getParentIndex(index);
```

```
33          }
34
35          // Store the new element in the vacant slot
36          elements.set(index, newElement);
37       }
38                                                                    746
39       /**                                                          747
40          Gets the minimum element stored in this heap.
41          @return the minimum element
42       */
43       public Comparable peek()
44       {
45          return elements.get(1);
46       }
47
48       /**
49          Removes the minimum element from this heap.
50          @return the minimum element
51       */
52       public Comparable remove()
53       {
54          Comparable minimum = elements.get(1);
55
56          // Remove last element
57          int lastIndex = elements.size() - 1;
58          Comparable last =
    elements.remove(lastIndex);
59
60          if (lastIndex > 1)
61          {
62             elements.set(1, last);
63             fixHeap();
64          }
65
66          return minimum;
67       }
68
69       /**
70          Turns the tree back into a heap, provided only the root
71          node violates the heap condition.
72       */
```

```
 73      private void fixHeap()
 74      {
 75         Comparable root = elements.get(1);
 76
 77         int lastIndex = elements.size() - 1;
 78         // Promote children of removed root while they are larger
    than last
 79
 80         int index = 1;
 81         boolean more = true;
 82         while (more)
 83         {
 84            int childIndex =
    getLeftChildIndex(index);
 85            if (childIndex <= lastIndex)
 86            {
 87               // Get smaller child
 88
 89               // Get left child first
 90               Comparable child =
    getLeftChild(index);
 91
 92               // Use right child instead if it is smaller
 93               if (getRightChildIndex(index) <=
    lastIndex
 94                     &&
    getRightChild(index).compareTo(child) < 0)
 95               {
 96                  childIndex =
    getRightChildIndex(index);
 97                  child = getRightChild(index);
 98               }
 99
100               // Check if larger child is smaller than root
101               if (child.compareTo(root) < 0)
102               {
103                  // Promote child
104                  elements.set(index, child);
105                  index = childIndex;
106               }
107               else
108               {
```

747 748

```
109                     // root is smaller than both children
110                     more = false;
111                  }
112               }
113               else
114               {
115                  // No children
116                  more = false;
117               }
118            }
119
120            // Store root element in vacant slot
121            elements.set(index, root);
122         }
123
124         /**
125            Returns the number of elements in this heap.
126         */
127         public int size()
128         {
129            return elements.size() - 1;
130         }
131
132         /**
133            Returns the index of the left child.
134            @param index the index of a node in this heap
135            @return the index of the left child of
the given node
136         */
137         private static int getLeftChildIndex(int
index)
138         {
139            return 2 * index;
140         }
141
142         /**
143            Returns the index of the right child.
144            @param index the index of a node in this heap
145            @return the index of the right child of the given node
146         */
```

```
147     private static int getRightChildIndex(int
index)
148     {
149        return 2 * index + 1;
150     }
151
152     /**
153        Returns the index of the parent.
154        @param index the index of a node in this heap
155        @return the index of the parent of the given node
156     */
157     private static int getParentIndex(int
index)
158     {
159        return index / 2;
160     }
161
162     /**
163        Returns the value of the left child.
164        @param index the index of a node in this heap
165        @return the value of the left child of the given node
166     */
167     private Comparable getLeftChild(int index)
168     {
169        return elements.get(2 * index);
170     }
171
172     /**
173        Returns the value of the right child.
174        @param index the index of a node in this heap
175        @return the value of the right child of the given node
176     */
177     private Comparable getRightChild(int index)
178     {
179        return elements.get(2 * index + 1);
180     }
181
182     /**
183        Returns the value of the parent.
184        @param index the index of a node in this heap
185        @return the value of the parent of the given node
```

```
186      */
187      private Comparable getParent(int index)
188      {
189         return elements.get(index / 2);
190      }
191
192      private ArrayList<Comparable> elements;
193   }
```

ch16/pqueue/HeapDemo.java

```
1   /**
2      This program demonstrates the use of a heap as a priority queue.
3   */
4   public class HeapDemo
5   {
6      public static void main(String[] args)
7      {
8         MinHeap q = new MinHeap();
9         q.add(new WorkOrder(3, "Shampoo
    carpets"));
10        q.add(new WorkOrder(7, "Empty trash"));
11        q.add(new WorkOrder(8, "Water plants"));
12        q.add(new WorkOrder(10, "Remove pencil
    sharpener shavings"));
13        q.add(new WorkOrder(6, "Replace light
    bulb"));
14        q.add(new WorkOrder(1, "Fix broken
    sink"));
15        q.add(new WorkOrder(9, "Clean coffee
    maker"));
16        q.add(new WorkOrder(2, "Order cleaning
    supplies"));
17
18        while (q.size() > 0)
19           System.out.println(q.remove());
20     }
21  }
```


ch16/pqueue/WorkOrder.java

```
1   /**
```

```
 2      This class encapsulates a work order with a priority.
 3   */
 4   public class WorkOrder implements Comparable
 5   {
 6      /**
 7         Constructs a work order with a given priority and description.
 8         @param aPriority the priority of this work order
 9         @param aDescription the description of this work order
10      */
11      public WorkOrder(int aPriority, String
aDescription)
12      {
13         priority = aPriority;
14         description = aDescription;
15      }
16
17      public String toString()
18      {
19         return "priority=" + priority + ",
description=" + description;
20      }
21
22      public int compareTo(Object otherObject)
23      {
24         WorkOrder other = (WorkOrder)
otherObject;
25         if (priority < other.priority) return -1;
26         if (priority > other.priority) return 1;
27         return 0;
28      }
29
30      private int priority;
31      private String description;
32   }
```

750

751

Output

```
priority=1, description=Fix broken sink
priority=2, description=Order cleaning supplies
priority=3, description=Shampoo carpets
priority=6, description=Replace light bulb
priority=7, description=Empty trash
```

```
priority=8, description=Water plants
priority=9, description=Clean coffee maker
priority=10, description=Remove pencil sharpener
shavings
```

SELF CHECK

15. The software that controls the events in a user interface keeps the events in a data structure. Whenever an event such as a mouse move or repaint request occurs, the event is added. Events are retrieved according to their importance. What abstract data type is appropriate for this application?

16. Could we store a binary search tree in an array so that we can quickly locate the children by looking at array locations `2 * index` and `2 * index + 1`?

16.10 The Heapsort Algorithm

Heaps are not only useful for implementing priority queues, they also give rise to an efficient sorting algorithm, heapsort. In its simplest form, the algorithm works as follows. First insert all elements to be sorted into the heap, then keep extracting the minimum.

The heapsort algorithm is based on inserting elements into a heap and removing them in sorted order.

This algorithm is an $O(n \log(n))$ algorithm: each insertion and removal is $O(\log(n))$, and these steps are repeated n times, once for each element in the sequence that is to be sorted.

Heapsort is an $O(n \log(n))$ algorithm.

The algorithm can be made a bit more efficient. Rather than inserting the elements one at a time, we will start with a sequence of values in an array. Of course, that array does not represent a heap. We will use the procedure of "fixing the heap" that you encountered in the preceding section as part of the element removal algorithm.

"Fixing the heap" operates on a binary tree whose child trees are heaps but whose root value may not be smaller than the descendants. The procedure turns the tree into a heap, by repeatedly promoting the smallest child value, moving the root value to its proper location.

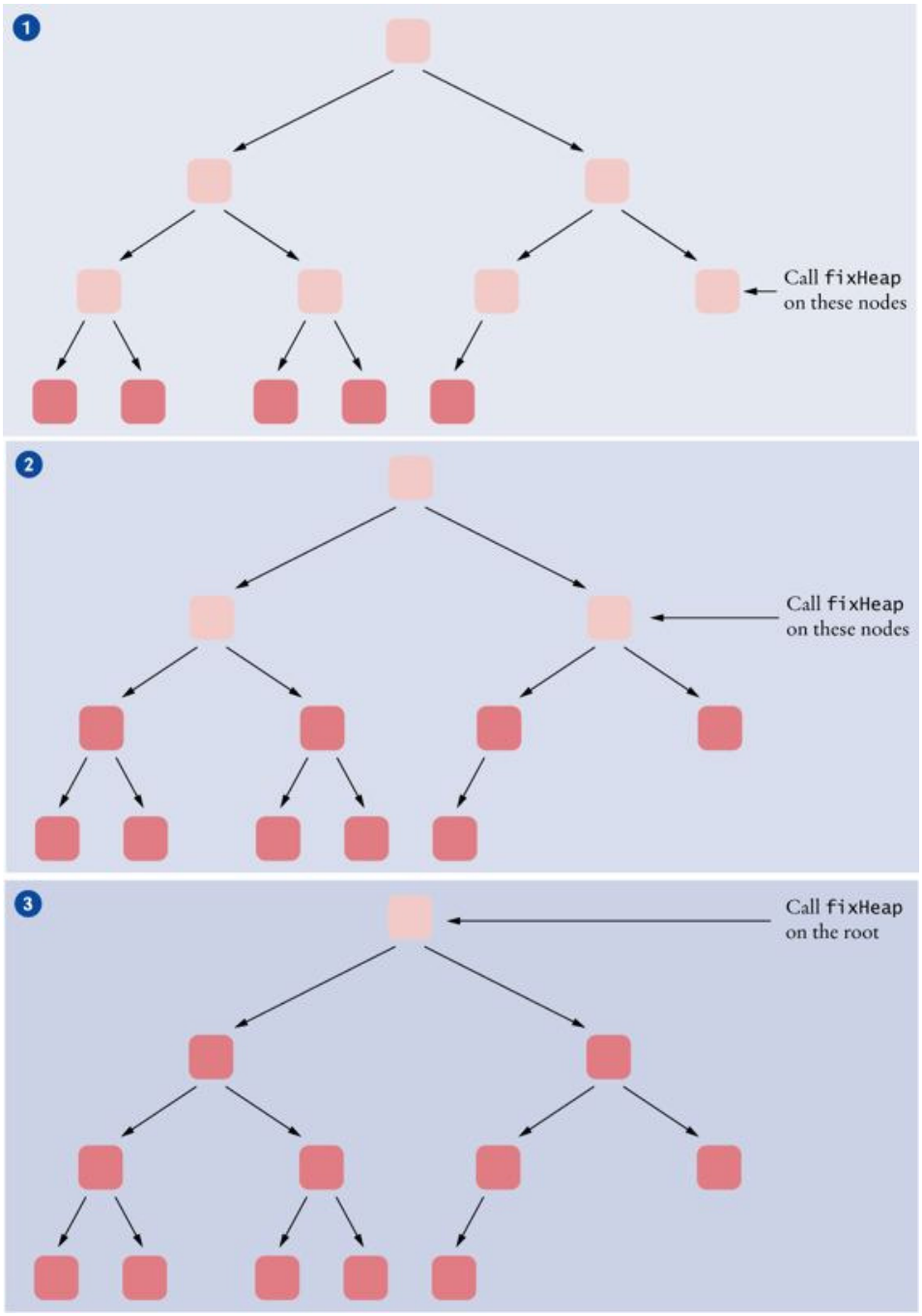
Of course, we cannot simply apply this procedure to the initial sequence of unsorted values—the child trees of the root are not likely to be heaps. But we can first fix small subtrees into heaps, then fix larger trees. Because trees of size 1 are automatically heaps, we can begin the fixing procedure with the subtrees whose roots are located in the next-to-lowest level of the tree.

The sorting algorithm uses a generalized `fixHeap` method that fixes a subtree with a given root index:

```
void fixHeap(int rootIndex, int lastIndex)
```

751

Figure 21



Turning a Tree into a Heap

Here, `lastIndex` is the index of the last node in the full tree. The `fixHeap` method needs to be invoked on all subtrees whose roots are in the next-to-last level. Then the subtrees whose roots are in the next level above are fixed, and so on. Finally, the fixup is applied to the root node, and the tree is turned into a heap (see Figure 21).

That repetition can be programmed easily. Start with the *last* node on the next-to-lowest level and work toward the left. Then go to the next higher level. The node index values then simply run backwards from the index of the last node to the index of the root.

```
int n = a.length - 1;
for (int i = (n - 1) / 2; i >= 0; i--)
   fixHeap(i, n);
```

Note that the loop ends with index 0. When working with a given array, we don't have the luxury of skipping the 0 entry. We consider the 0 entry the root and adjust the formulas for computing the child and parent index values.

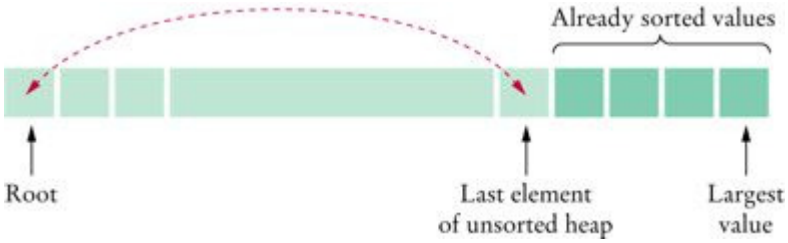
After the array has been turned into a heap, we repeatedly remove the root element. Recall from the preceding section that removing the root element is achieved by placing the last element of the tree in the root and calling the `fixHeap` method.

Rather than moving the root element into a separate array, we will *swap* the root element with the last element of the tree and then reduce the tree length. Thus, the removed root ends up in the last position of the array, which is no longer needed by the heap. In this way, we can use the same array both to hold the heap (which gets shorter with each step) and the sorted sequence (which gets longer with each step).

There is just a minor inconvenience. When we use a min-heap, the sorted sequence is accumulated in reverse order, with the smallest element at the end of the array. We could reverse the sequence after sorting is complete. However, it is easier to use a max-heap rather than a min-heap in the heapsort algorithm. With this modification, the largest value is placed at the end of the array after the first step. After the next step, the next-largest value is swapped from the heap root to the second position from the end, and so on (see Figure 22).

The following class implements the heapsort algorithm.

Figure 22



Using Heapsort to Sort an Array

753

754

ch16/heapsort/HeapSorter.java

```
1  /**
2     This class applies the heapsort algorithm to sort an array.
3  */
4  public class HeapSorter
5  {
6     /**
7        Constructs a heap sorter that sorts a given array.
8        @param anArray an array of integers
9     */
10    public HeapSorter(int[] anArray)
11    {
12       a = anArray;
13    }
14
15    /**
16       Sorts the array managed by this heap sorter.
17    */
18    public void sort()
19    {
20       int n = a.length - 1;
21       for (int i = (n - 1) / 2; i >= 0; i--)
22          fixHeap(i, n);
23       while (n > 0)
24       {
25          swap(0, n);
26          n--;
```

```
27            fixHeap(0, n);
28         }
29      }
30
31      /**
32         Ensures the heap property for a subtree, provided its
33         children already fulfill the heap property.
34         @param rootIndex the index of the subtree to be fixed
35         @param lastIndex the last valid index of the tree that
36         contains the subtree to be fixed
37      */
38      private void fixHeap(int rootIndex, int
lastIndex)
39      {
40         // Remove root
41         int rootValue = a[rootIndex];
42
43         // Promote children while they are larger than the root
44
45         int index = rootIndex;
46         boolean more = true;
47         while (more)
48         {
49            int childIndex =
getLeftChildIndex(index);
50            if (childIndex <= lastIndex)
51            {
52               // Use right child instead if it is larger
53               int rightChildIndex =
getRightChildIndex(index);
54               if (rightChildIndex <= lastIndex
55                     && a[rightChildIndex] >
a[childIndex])
56               {
57                  childIndex = rightChildIndex;
58               }
59
60               if (a[childIndex] > rootValue)
61               {
62                  // Promote child
63                  a[index] = a[childIndex];
```

754

755

```
64                      index = childIndex;
65                   }
66                   else
67                   {
68                      // Root value is larger than both children
69                      more = false;
70                   }
71                }
72                else
73                {
74                   // No children
75                   more = false;
76                }
77             }
78
79             // Store root value in vacant slot
80             a[index] = rootValue;
81          }
82
83          /**
84             Swaps two entries of the array.
85             @param i the first position to swap
86             @param j the second position to swap
87          */
88          private void swap(int i, int j)
89          {
90             int temp = a[i];
91             a[i] = a[j];
92             a[j] = temp;
93          }
94
95          /**
96             Returns the index of the left child.
97             @param index the index of a node in this heap
98             @return the index of the left child of the given node
99          */
100         private static int getLeftChildIndex(int
    index)
101         {
102            return 2 * index + 1;
103         }
```

```
104
105     /**
106        Returns the index of the right child.
107        @param index the index of a node in this heap
108        @return the index of the right child of the given node
109     */
110     private static int getRightChildIndex(int
index)
111     {
112        return 2 * index + 2;
113     }
114
115     private int[] a;
116  }
```

755 756

SELF CHECK

17. Which algorithm requires less storage, heapsort or merge sort?

18. Why are the computations of the left child index and the right child index in the `HeapSorter` different than in `MinHeap`?

CHAPTER SUMMARY

1. A set is an unordered collection of distinct elements. Elements can be added, located, and removed.
2. Sets don't have duplicates. Adding a duplicate of an element that is already present is silently ignored.
3. The `HashSet` and `TreeSet` classes both implement the `Set` interface.
4. An iterator visits all elements in a set.
5. A set iterator does not visit the elements in the order in which you inserted them. The set implementation rearranges the elements so that it can locate them quickly.
6. You cannot add an element to a set at an iterator position.
7. A map keeps associations between key and value objects.

8. The `HashMap` and `TreeMap` classes both implement the `Map` interface.
9. To find all keys and values in a map, iterate through the key set and find the values that correspond to the keys.
10. A hash function computes an integer value from an object.
11. A good hash function minimizes *collisions*—identical hash codes for different objects.

756

757

12. A hash table can be implemented as an array of *buckets*—sequences of nodes that hold elements with the same hash code.
13. If there are no or only a few collisions, then adding, locating, and removing hash table elements takes constant or $O(1)$ time.
14. The table size should be a prime number, larger than the expected number of elements.
15. Define `hashCode` methods for your own classes by combining the hash codes for the instance variables.
16. Your `hashCode` method must be compatible with the `equals` method.
17. In a hash map, only the keys are hashed.
18. A binary tree consists of nodes, each of which has at most two child nodes.
19. All nodes in a binary search tree fulfill the property that the descendants to the left have smaller data values than the node data value, and the descendants to the right have larger data values.
20. When removing a node with only one child from a binary search tree, the child replaces the node to be removed.
21. When removing a node with two children from a binary search tree, replace it with the smallest node of the right subtree.
22. If a binary search tree is balanced, then adding an element takes $O(\log(n))$ time.

23. Tree traversal schemes include preorder traversal, inorder traversal, and postorder traversal.

24. Postorder traversal of an expression tree yields the instructions for evaluating the expression on a stack-based calculator.

25. The `TreeSet` class uses a form of balanced binary trees that guarantees that adding and removing an element takes $O(\log(n))$ time.

26. To use a tree set, the elements must be comparable.

27. When removing an element from a priority queue, the element with the highest priority is retrieved.

28. A heap is an almost complete tree in which the values of all nodes are at most as large as those of their descendants.

29. Inserting or removing a heap element is an $O(\log(n))$ operation.

30. The regular layout of a heap makes it possible to store heap nodes efficiently in an array.

31. The heapsort algorithm is based on inserting elements into a heap and removing them in sorted order.

32. Heapsort is an $O(n \log(n))$ algorithm.

757

758

CLASSES, OBJECTS, AND METHODS INTRODUCED IN THIS CHAPTER

```
java.util.Collection<E>
   contains
   remove
   size
java.util.HashMap<K, V>
java.util.HashSet<K, V>
java.util.Map<K, V>
   get
   keySet
   put
   remove
```

```
java.util.PriorityQueue<E>
   remove
java.util.Set<E>
java.util.TreeMap<K, V>
java.util.TreeSet<K, V>
```

FURTHER READING

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, 2nd edition, MIT Press, 2001.

REVIEW EXERCISES

★ **Exercise R16.1.** What is the difference between a set and a map?

★ **Exercise R16.2.** What implementations does the Java library provide for the abstract set type?

★★ **Exercise R16.3.** What are the fundamental operations on the abstract set type? What additional methods does the `Set` interface provide? (Look up the interface in the API documentation.)

★★ **Exercise R16.4.** The union of two sets *A* and *B* is the set of all elements that are contained in *A, B*, or both. The intersection is the set of all elements that are contained in *A* and *B*. How can you compute the union and intersection of two sets, using the four fundamental set operations described on page 701?

★★ **Exercise R16.5.** How can you compute the union and intersection of two sets, using some of the methods that the `java.util.Set` interface provides? (Look up the interface in the API documentation.)

★ **Exercise R16.6.** Can a map have two keys with the same value? Two values with the same key?

★ **Exercise R16.7.** A map can be implemented as a set of (*key*, *value*) pairs. Explain.

★★ **Exercise R16.8.** When implementing a map as a hash set of (*key*, *value*) pairs, how is the hash code of a pair computed?

★ **Exercise R16.9.** Verify the hash codes of the strings `"Jim"` and `"Joe"` in Table 1.

★ **Exercise R16.10.** From the hash codes in Table 1, show that Figure 6 accurately shows the locations of the strings if the hash table size is 101.

★ **Exercise R16.11.** What is the difference between a binary tree and a binary search tree? Give examples of each.

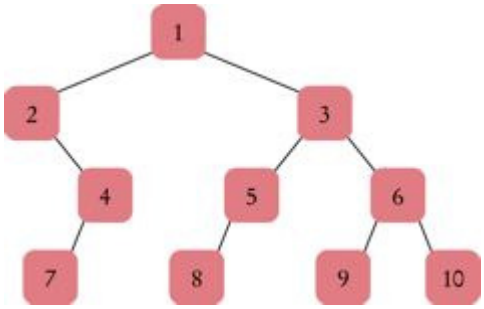
★ **Exercise R16.12.** What is the difference between a balanced tree and an unbalanced tree? Give examples of each.

★ **Exercise R16.13.** The following elements are inserted into a binary search tree. Make a drawing that shows the resulting tree after each insertion.

```
Adam
Eve
Romeo
Juliet
Tom
Dick
Harry
```

★★ **Exercise R16.14.** Insert the elements of Exercise R16.13 in opposite order. Then determine how the `BinarySearchTree.print` method prints out both the tree from Exercise R16.13 and this tree. Explain how the printouts are related.

★★ **Exercise R16.15.** Consider the following tree. In which order are the nodes printed by the `BinarySearchTree.print` method?



★★ **Exercise R16.16.** Could a priority queue be implemented efficiently as a binary search tree? Give a detailed argument for your answer.

★★★ **Exercise R16.17.** Will preorder, inorder, or postorder traversal print a heap in sorted order? Why or why not?

759

760

★★★ **Exercise R16.18.** Prove that a heap of height h contains at least 2^{h-1} elements but less than 2^h elements.

★★★ **Exercise R16.19.** Suppose the heap nodes are stored in an array, starting with index 1. Prove that the child nodes of the heap node with index i have index $2 \cdot i$ and $2 \cdot i + 1$, and the parent heap node of the node with index i has index $i/2$.

★★ **Exercise R16.20.** Simulate the heapsort algorithm manually to sort the array

```
11 27 8 14 45 6 24 81 29 33
```

Show all steps.

Additional review exercises are available in WileyPLUS.

PROGRAMMING EXERCISES

★ **Exercise P16.1.** Write a program that reads text from `System.in` and breaks it up into individual words. Insert the words into a tree set. At the end of the input file, print all words, followed by the size of the resulting set. This program determines how many unique words a text file has.

★ **Exercise P16.2.** Insert the 13 standard colors that the `Color` class predefines (that is, `Color.PINK`, `Color.GREEN`, and so on) into a set. Prompt the user to enter a color by specifying red, green, and blue integer values between 0 and 255. Then tell the user whether the resulting color is in the set.

★★★ **Exercise P16.3.** Add a `debug` method to the `HashSet` implementation in Section 16.3 that prints the nonempty buckets of the hash table. Run

the test program at the end of Section 16.3. Call the `debug` method after all additions and removals and verify that Figure 6 accurately represents the state of the hash table.

★★ **Exercise P16.4.** Write a program that keeps a map in which both keys and values are strings—the names of students and their course grades. Prompt the user of the program to add or remove students, to modify grades, or to print all grades. The printout should be sorted by name and formatted like this:

```
Carl: B+
Joe: C
Sarah: A
```

★★★ **Exercise P16.5.** Reimplement Exercise P16.4 so that the keys of the map are objects of class `Student`. A student should have a first name, a last name, and a unique integer ID. For grade changes and removals, lookup should be by ID. The printout should be sorted by last name. If two students have the same last name, then use the first name as tie breaker. If the first names are also identical, then use the integer ID. *Hint*: Use two maps.

760

761

★★ **Exercise P16.6.** Supply compatible `hashCode` and `equals` methods to the `Student` class described in Exercise P16.5. Test the hash code by adding `Student` objects to a hash set.

★ **Exercise P16.7.** Supply compatible `hashCode` and `equals` methods to the `BankAccount` class of Chapter 7. Test the `hashCode` method by printing out hash codes and by adding `BankAccount` objects to a hash set.

★★ **Exercise P16.8.** Design an `IntTree` class that stores only integers, not objects. Support the same methods as the `BinarySearchTree` class in the book.

★★ **Exercise P16.9.** Design a data structure `IntSet` that can hold a set of integers. Hide the private implementation: a binary search tree of `Integer` objects. Provide the following methods:

- A constructor to make an empty set

- `void add(int x)` to add `x` if it is not present
- `void remove(int x)` to remove `x` if it is present
- `void print()` to print all elements currently in the set
- `boolean find(int x)` to test whether `x` is present

★★ **Exercise P16.10.** Reimplement the set class from Exercise P16.9 by using a `TreeSet<Integer>`. In addition to the methods specified in Exercise P16.9, supply an `iterator` method yielding an object that supports *only* the `hasNext/next` methods.

The `next` method should return an `int`, not an object. For that reason, you cannot simply return the iterator of the tree set.

★ **Exercise P16.11.** Reimplement the set class from Exercise P16.9 by using a `TreeSet<Integer>`. In addition to the methods specified in Exercise P16.9, supply methods

```
IntSet union(IntSet other)
IntSet intersection(IntSet other)
```

that compute the union and intersection of two sets.

★★ **Exercise P16.12.** Implement the *sieve of Eratosthenes:* a method for computing prime numbers, known to the ancient Greeks. Choose an n. This method will compute all prime numbers up to n. First insert all numbers from 2 to n into a set. Then erase all multiples of 2 (except 2); that is, 4, 6, 8, 10, 12, …. Erase all multiples of 3; that is, 6, 9, 12, 15, …. Go up to $\sqrt{n}$. Then print the set.

★ **Exercise P16.13.** Write a method of the `BinarySearchTree` class

```
Comparable smallest()
```

that returns the smallest element of a tree. You will also need to add a method to the `Node` class.

★★★ **Exercise P16.14.** Change the `BinarySearchTree.print` method to print the tree as a tree shape. You can print the tree sideways. Extra credit if you instead display the tree with the root node centered on the top.

761

762

★ **Exercise P16.15.** Implement methods that use preorder and postorder traversal to print the elements in a binary search tree.

★★★ **Exercise P16.16.** In the `BinarySearchTree` class, modify the `remove` method so that a node with two children is replaced by the largest child of the left subtree.

★★ **Exercise P16.17.** Suppose an interface `Visitor` has a single method

```
void visit(Object obj)
```

Supply methods

```
void inOrder(Visitor v)
void preOrder(Visitor v)
void postOrder(Visitor v)
```

to the `BinarySearchTree` class. These methods should visit the tree nodes in the specified traversal order and apply the `visit` method to the data of the visited node.

★★ **Exercise P16.18.** Apply Exercise P16.17 to compute the average value of the elements in a binary search tree filled with `Integer` objects. That is, supply an object of an appropriate class that implements the `Visitor` interface.

★★ **Exercise P16.19.** Modify the implementation of the `MinHeap` class so that the parent and child index positions and elements are computed directly, without calling helper methods.

★★★ **Exercise P16.20.** Modify the implementation of the `MinHeap` class so that the 0 element of the array is not wasted.

★ **Exercise P16.21.** Time the results of heapsort and merge sort. Which algorithm behaves better in practice?

Additional programming exercises are available in WileyPLUS.

PROGRAMMING PROJECTS

★★★ **Project 16.1.** Implement a `BinaryTreeSet` class that uses a `TreeSet` to store its elements. You will need to implement an iterator that iterates through the nodes in sorted order. This iterator is somewhat complex, because sometimes you need to backtrack. You can either add a reference to the parent node in each `Node` object, or have your iterator object store a stack of the visited nodes.

★★★ **Project 16.2.** Implement an expression evaluator that uses a parser to build an expression tree, such as in Section 16.6. (Note that the resulting tree is a binary tree but not a binary search tree.) Then use postorder traversal to evaluate the expression, using a stack for the intermediate results.

★★★ **Project 16.3.** Program an animation of the heapsort algorithm, displaying the tree graphically and stopping after each call to `fixHeap`.

762

763

ANSWERS TO SELF-CHECK QUESTIONS

1. Efficient set implementations can quickly test whether a given element is a member of the set.
2. Sets do not have an ordering, so it doesn't make sense to add an element at a particular iterator position, or to traverse a set backwards.
3. A set stores elements. A map stores associations between keys and values.
4. The ordering does not matter, and you cannot have duplicates.
5. Yes, the hash set will work correctly. All elements will be inserted into a single bucket.

6. It locates the next bucket in the bucket array and points to its first element.

7. $31 \times 116 + 111 = 3707$.

8. 13.

9. In a tree, each node can have any number of children. In a binary tree, a node has at most two children. In a balanced binary tree, all nodes have approximately as many descendants to the left as to the right.

10. For example, `Sarah`. Any string between `Romeo` and `Tom` will do.

11. For both trees, the inorder traversal is `3 + 4 * 5`.

12. No—for example, consider the children of +. Even without looking up the Unicode codes for `3`, `4`, and +, it is obvious that + isn't between `3` and `4`.

13. When it is desirable to visit the set elements in sorted order.

14. No—it would never be able to tell two coins apart. Thus, it would think that all coins are duplicates of the first.

15. A priority queue is appropriate because we want to get the important events first, even if they have been inserted later.

16. Yes, but a binary search tree isn't almost filled, so there may be holes in the array. We could indicate the missing nodes with `null` elements.

17. Heapsort requires less storage because it doesn't need an auxiliary array.

18. The `MinHeap` wastes the 0 entry to make the formulas more intuitive. When sorting an array, we don't want to waste the 0 entry, so we adjust the formulas instead.