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.
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.

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.

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).

![Figure 2: Set Classes and Interfaces in the Standard Library](bj3ch16.fm Page 701 Monday, January 29, 2007 9:36 PM)
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.

```java
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:

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

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

```java
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.

```java
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:

```java
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.

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.

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.
16.1 Sets

ch16/set/SetDemo.java

```java
import java.util.HashSet;
import java.util.Scanner;
import java.util.Set;

/**
   * This program demonstrates a set of strings. The user can add and remove strings.
   */
public class SetDemo
{
    public static void main(String[] args)
    {
        Set<String> names = new HashSet<String>();
        Scanner in = new Scanner(System.in);

        boolean done = false;
        while (!done)
        {
            System.out.print("Add name, Q when done: ");
            String input = in.next();
            if (input.equalsIgnoreCase("Q"))
                done = true;
            else
            {
                names.add(input);
                print(names);
            }
        } 
        done = false;
        while (!done)
        {
            System.out.print("Remove name, Q when done: ");
            String input = in.next();
            if (input.equalsIgnoreCase("Q"))
                done = true;
            else
            {
                names.remove(input);
                print(names);
            }
        }
    }
    /**
     * Prints the contents of a set of strings.
     * @param s a set of strings
     */
    private static void print(Set<String> s)
    {
        System.out.print("{ ");
        for (String element : s)
        {
            System.out.print(element + ", ");
        }
        System.out.println("} ");
    }
}
```
CHAPTER 16 • Advanced Data Structures

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
16.2 Maps

A map keeps associations between key and value objects. Figure 3 gives a typical example: a map that associates names with colors. This map might describe the favorite colors of various people. 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).

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

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

Figure 3 A Map
Use the `put` method to add an association:

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

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

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

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

```java
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:

```java
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`:

```java
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.

```
ch16/map/MapDemo.java
```

```java
import java.awt.Color;
import java.util.HashMap;
import java.util.Map;
import java.util.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

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

computes the hash code of the object `x`.

---

### 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?

---

```java
/**
 * This program demonstrates a map that maps names to colors.
 */
public class MapDemo {
    public static void main(String[] args) {
        Map<String, Color> favoriteColors = new HashMap<String, Color>();
        favoriteColors.put("Juliet", Color.PINK);
        favoriteColors.put("Romeo", Color.GREEN);
        favoriteColors.put("Adam", Color.BLUE);
        favoriteColors.put("Eve", Color.PINK);
        Set<String> keySet = favoriteColors.keySet();
        for (String key : keySet) {
            Color value = favoriteColors.get(key);
            System.out.println(key + "->" + value);
        }
    }
}
```

**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]
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.

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).

---

Table 1  Sample Strings and Their Hash Codes

<table>
<thead>
<tr>
<th>String</th>
<th>Hash Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Adam&quot;</td>
<td>2035631</td>
</tr>
<tr>
<td>&quot;Eve&quot;</td>
<td>70068</td>
</tr>
<tr>
<td>&quot;Harry&quot;</td>
<td>69496448</td>
</tr>
<tr>
<td>&quot;Jim&quot;</td>
<td>74478</td>
</tr>
<tr>
<td>&quot;Joe&quot;</td>
<td>74656</td>
</tr>
<tr>
<td>&quot;Juliet&quot;</td>
<td>-2065036585</td>
</tr>
<tr>
<td>&quot;Katherine&quot;</td>
<td>2079199209</td>
</tr>
<tr>
<td>&quot;Sue&quot;</td>
<td>83491</td>
</tr>
</tbody>
</table>

---

A good hash function minimizes collisions—identical hash codes for different objects.
16.3 Hash Tables

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:

```java
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*.

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.

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

![A Hash Table with Buckets to Store Elements with the Same Hash Code](image)

*Figure 6* A Hash Table with Buckets to Store Elements with the Same Hash Code
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.

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.

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.

```java
import java.util.AbstractSet;
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * A hash set stores an unordered collection of objects, using
 * a hash table.
 */
public class HashSet extends AbstractSet {

    /**
     * Constructs a hash table.
     * @param bucketsLength the length of the buckets array
     */
    public HashSet(int bucketsLength) {
        buckets = new Node[bucketsLength];
        size = 0;
    }
```
16.3 Hash Tables

```
/**
 * Tests for set membership.
 * @param x an object
 * @return true if x is an element of this set
 */
public boolean contains(Object x) {
    int h = x.hashCode();
    if (h < 0) h = -h;
    h = h % buckets.length;
    Node current = buckets[h];
    while (current != null) {
        if (current.data.equals(x)) return true;
        current = current.next;
    }
    return false;
}

/**
 * Adds an element to this set.
 * @param x an object
 * @return true if x is a new object, false if x was already in the set
 */
public boolean add(Object x) {
    int h = x.hashCode();
    if (h < 0) h = -h;
    h = h % buckets.length;
    Node current = buckets[h];
    while (current != null) {
        if (current.data.equals(x)) return false; // Already in the set
        current = current.next;
    }
    Node newNode = new Node();
    newNode.data = x;
    newNode.next = buckets[h];
    buckets[h] = newNode;
    size++;
    return true;
}

/**
 * Removes an object from this set.
 * @param x an object
 * @return true if x was removed from this set, false if x was not an element of this set
 */
```
public boolean remove(Object x) {
    int h = x.hashCode();
    if (h < 0) h = -h;
    h = h % buckets.length;

    Node current = buckets[h];
    Node previous = null;
    while (current != null) {
        if (current.data.equals(x)) {
            if (previous == null) buckets[h] = current.next;
            else previous.next = current.next;
            size--;
            return true;
        }
        previous = current;
        current = current.next;
    }
    return false;
}

/**
   * Returns an iterator that traverses the elements of this set.
   * @return a hash set iterator
   */
public Iterator iterator() {
    return new HashSetIterator();
}

/**
   * Gets the number of elements in this set.
   * @return the number of elements
   */
public int size() {
    return size;
}

private Node[] buckets;
private int size;

private class Node {
    public Object data;
    public Node next;
}

private class HashSetIterator implements Iterator {
    /**
     * Constructs a hash set iterator that points to the first element of the hash set.
     */

/*
 * HashSetIterator
 * 
 * public HashSetIterator()
 * {
 *     current = null;
 *     bucket = -1;
 *     previous = null;
 *     previousBucket = -1;
 * }
 *
 * public boolean hasNext()
 * {
 *     if (current != null && current.next != null)
 *         return true;
 *     for (int b = bucket + 1; b < buckets.length; b++)
 *         if (buckets[b] != null) return true;
 *     return false;
 * }
 *
 * public Object next()
 * {
 *     previous = current;
 *     previousBucket = bucket;
 *     if (current == null || current.next == null)
 *     {
 *         // Move to next bucket
 *         bucket++;
 *         while (bucket < buckets.length
 *              && buckets[bucket] == null)
 *             bucket++;
 *         if (bucket < buckets.length)
 *             current = buckets[bucket];
 *         else
 *             throw new NoSuchElementException();
 *     } else // Move to next element in bucket
 *     current = current.next;
 *     return current.data;
 * }
 *
 * public void remove()
 * {
 *     if (previous != null && previous.next == current)
 *         previous.next = current.next;
 *     else if (previousBucket < bucket)
 *         buckets[bucket] = current.next;
 *     else
 *         throw new IllegalStateException();
 *     current = previous;
 *     bucket = previousBucket;
 * }
 *
 * private int bucket;
 * private Node current;
private int previousBucket;
private Node previous;
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:

```java
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.

```java
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.

```java
class Coin
{ public int hashCode()
{ int h1 = name.hashCode();
    int h2 = new Double(value).hashCode();
    ...
```
Then combine the two hash codes.

```java
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:

```java
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.

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.

```java
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
16.4 Computing Hash Codes

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.

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.

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

```java
/**
 * A coin with a monetary value.
 */
public class Coin {
    /**
     * Constructs a coin.
     * @param aValue the monetary value of the coin
     * @param aName the name of the coin
     */
    public Coin(double aValue, String aName) {
        value = aValue;
        name = aName;
    }

    /**
     * Gets the coin value.
     * @return the value
     */
    public double getValue() {
        return value;
    }
}
```
/**
 * Gets the coin name.
 * @return the name
 */

public String getName()
{
    return name;
}

public boolean equals(Object otherObject)
{
    if (otherObject == null) return false;
    if (getClass() != otherObject.getClass()) return false;
    Coin other = (Coin) otherObject;
    return value == other.value && name.equals(other.name);
}

public int hashCode()
{
    int h1 = name.hashCode();
    int h2 = new Double(value).hashCode();
    final int HASH_MULTIPLIER = 29;
    int h = HASH_MULTIPLIER * h1 + h2;
    return h;
}

public String toString()
{
    return "Coin[value=" + value + ",name=" + name + "]";
}

private double value;
private String name;

ch16/hashcode/CoinHashCodePrinter.java

/**
 * A program that prints hash codes of coins.
 */

public class CoinHashCodePrinter
{
    public static void main(String[] args)
    {
        Coin coin1 = new Coin(0.25, "quarter");
        Coin coin2 = new Coin(0.25, "quarter");
        Coin coin3 = new Coin(0.05, "nickel");
        System.out.println("hash code of coin1=
            + coin1.hashCode());
### 16.4 Computing Hash Codes

Output

```java
System.out.println("hash code of coin1=");
System.out.println("hash code of coin2=");
System.out.println("hash code of coin3=");
```

```java
Set<Coin> coins = new HashSet<Coin>();
coins.add(coin1);
coins.add(coin2);
coins.add(coin3);
```

```java
for (Coin c : coins)
    System.out.println(c);
```

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:

```java
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.
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.

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”. 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.

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.
16.5 Binary Search Trees

Figure 7 A Binary Search Tree

Figure 8 A Binary Tree That Is Not a Binary Search Tree
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
For example, consider the tree in Figure 9. It is the result of the following statements:

```java
BinarySearchTree tree = new BinarySearchTree();
   tree.add("Juliet");  // 1
   tree.add("Tom");    // 2
   tree.add("Dick");   // 3
   tree.add("Harry");  // 4

We want to insert a new element Romeo into it.

   tree.add("Romeo");  // 5
```

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

```java
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:

```java
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).
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.

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).

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
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 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.

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].

**Figure 13** An Unbalanced Binary Search Tree

```java
/**
 * This class implements a binary search tree whose
 * nodes hold objects that implement the Comparable
 * interface.
 */
public class BinarySearchTree {

```
```java
/**
 * Constructs an empty tree.
 */
public BinarySearchTree()
{
    root = null;
}

/**
 * Inserts a new node into the tree.
 * @param obj the object to insert
 */
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);
}

/**
 * Tries to find an object in the tree.
 * @param obj the object to find
 * @return true if the object is contained in the tree
 */
public boolean find(Comparable obj)
{
    Node current = root;
    while (current != null)
    {
        int d = current.data.compareTo(obj);
        if (d == 0) return true;
        else if (d > 0) current = current.left;
        else current = current.right;
    }
    return false;
}

/**
 * Tries to remove an object from the tree. Does nothing
 * if the object is not contained in the tree.
 * @param obj the object to remove
 */
public void remove(Comparable obj)
{
    // Find node to be removed
    Node toBeRemoved = root;
    Node parent = null;
    boolean found = false;
    while (!found && toBeRemoved != null)
    {
```
```
```java
int d = toBeRemoved.data.compareTo(obj);
if (d == 0) found = true;
else {
    parent = toBeRemoved;
    if (d > 0) toBeRemoved = toBeRemoved.left;
    else toBeRemoved = toBeRemoved.right;
}
}

if (!found) return;

// toBeRemoved contains obj
// If one of the children is empty, use the other
if (toBeRemoved.left == null || toBeRemoved.right == null) {
    Node newChild;
    if (toBeRemoved.left == null)
        newChild = toBeRemoved.right;
    else
        newChild = toBeRemoved.left;
    if (parent == null) // Found in root
        root = newChild;
    else if (parent.left == toBeRemoved)
        parent.left = newChild;
    else
        parent.right = newChild;
    return;
}

// Neither subtree is empty
// Find smallest element of the right subtree
Node smallestParent = toBeRemoved;
Node smallest = toBeRemoved.right;
while (smallest.left != null)
    { smallestParent = smallest;
    smallest = smallest.left;
    }

// smallest contains smallest child in right subtree
// Move contents, unlink child
toBeRemoved.data = smallest.data;
smallestParent.left = smallest.right;
```
 CHAPTER 16  •  Advanced Data Structures

```java
/**
   * Prints the contents of the tree in sorted order.
   */
public void print()
{
    if (root != null)
    root.printNodes();
    System.out.println();
}

private Node root;

/**
   * A node of a tree stores a data item and references to the child nodes to the left and to the right.
   */
private class Node
{
    /**
     * Inserts a new node as a descendant of this node.
     * @param newNode the node to insert
     */
    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 (right == null) right = newNode;
             else right.addNode(newNode);
       }
    }

    /**
     * Prints this node and all of its descendants in sorted order.
     */
    public void printNodes()
    {
        if (left != null)
        left.printNodes();
        System.out.println(data + " ");
        if (right != null)
        right.printNodes();
    }
}

public Comparable data;
public Node left;
public Node right;
```
16.6  Tree Traversal

**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.

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
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:

```java
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.

```java
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*.

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

\[(3 + 4) \times 5\]
\[3 + 4 \times 5\]

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. This observation yields an algorithm for evaluating arithmetic expressions. First, turn the expression into a tree. Then carry out a postorder traversal of the
734 CHAPTER 16 • Advanced Data Structures

expression tree and apply the operations in the given order. The result is the value of the expression.

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,

<table>
<thead>
<tr>
<th>Standard Notation</th>
<th>Łukasiewicz Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 + 4</td>
<td>+ 3 4</td>
</tr>
<tr>
<td>3 + 4 * 5</td>
<td>+ 3 * 4 5</td>
</tr>
<tr>
<td>3 * (4 + 5)</td>
<td>* 3 + 4 5</td>
</tr>
<tr>
<td>(3 + 4) * 5</td>
<td>* + 3 4 5</td>
</tr>
<tr>
<td>3 + 4 + 5</td>
<td>+ + 3 4 5</td>
</tr>
</tbody>
</table>

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.

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.

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

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

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.

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 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.
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,

```java
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

```
Output

<table>
<thead>
<tr>
<th>output</th>
<th>expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td>0.25</td>
<td>0.25</td>
</tr>
</tbody>
</table>

SELF CHECK

13. When would you choose a tree set over a hash set?
14. Suppose we define a coin comparator whose compareTo 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.
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.

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:

```java
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.
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.

![Figure 16 An Almost Complete Tree](image-url)
16.9  Heaps

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 17](A Heap)

![Figure 18](Inserting an Element into a Heap)
Figure 18 (continued)  Inserting an Element into a Heap

1. **Demote parents**

2. **Insert element into vacant slot**
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.
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,
16.9 Heaps

where \( b \) is the height of the tree. A heap of height \( b \) contains at least \( 2^{b-1} \) elements, but less than \( 2^b \) elements. In other words, if \( n \) is the number of elements, then

\[
2^{b-1} \leq n < 2^b
\]

or

\[
b - 1 \leq \log_2(n) < b
\]

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

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.

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
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.

```java
import java.util.*;

/**
 * This class implements a heap.
 */
public class MinHeap {
    /**
     * Constructs an empty heap.
     */
    public MinHeap() {
        elements = new ArrayList<Comparable>();
        elements.add(null);
    }

    /**
     * Adds a new element to this heap.
     * @param newElement the element to add
     */
    public void add(Comparable newElement) {
        // Add a new leaf
        elements.add(null);
        int index = elements.size() - 1;

        // Demote parents that are larger than the new element
        while (index > 1 && getParent(index).compareTo(newElement) > 0) {
            elements.set(index, getParent(index));
            index = getParentIndex(index);
        }

        // Store the new element in the vacant slot
        elements.set(index, newElement);
    }
}
### 16.9 Heaps

```java
/**
 * Gets the minimum element stored in this heap.
 * @return the minimum element
 */
public Comparable peek()
{
    return elements.get(1);
}

/**
 * Removes the minimum element from this heap.
 * @return the minimum element
 */
public Comparable remove()
{
    Comparable minimum = elements.get(1);

    // Remove last element
    int lastIndex = elements.size() - 1;
    Comparable last = elements.remove(lastIndex);
    if (lastIndex > 1)
    {
        elements.set(1, last);
        fixHeap();
    }

    return minimum;
}

/**
 * Turns the tree back into a heap, provided only the root
 * node violates the heap condition.
 */
private void fixHeap()
{
    Comparable root = elements.get(1);
    int lastIndex = elements.size() - 1;
    // Promote children of removed root while they are larger than last
    int index = 1;
    boolean more = true;
    while (more)
    {
        int childIndex = getLeftChildIndex(index);
        if (childIndex <= lastIndex)
        {
            // Get smaller child
            Comparable child = getLeftChild(index);
            // Get left child first
            Comparable child = getLeftChild(index);
        }
    }
```
Use right child instead if it is smaller

```
if (getRightChildIndex(index) <= lastIndex
    && getRightChild(index).compareTo(child) < 0)
{
    childIndex = getRightChildIndex(index);
    child = getRightChild(index);
}
```

Check if larger child is smaller than root

```
if (child.compareTo(root) < 0)
{
    // Promote child
    elements.set(index, child);
    index = childIndex;
}
else
{
    // root is smaller than both children
    more = false;
}
else
{
    // No children
    more = false;
}
```

Store root element in vacant slot

```
elements.set(index, root);
```

// Returns the number of elements in this heap.
```
public int size()
{
    return elements.size() - 1;
}
```

// Returns the index of the left child.
```
    @param index the index of a node in this heap
    @return the index of the left child of the given node
```
```
private static int getLeftChildIndex(int index)
{
    return 2 * index;
}
```

// Returns the index of the right child.
```
    @param index the index of a node in this heap
    @return the index of the right child of the given node
```
```
private static int getRightChildIndex(int index)
{
    return 2 * index + 1;
}
```
### Heaps

```java
private static int getRightChildIndex(int index) {
    return 2 * index + 1;
}

/**
 * Returns the index of the parent.
 * @param index the index of a node in this heap
 * @return the index of the parent of the given node
 */
private static int getParentIndex(int index) {
    return index / 2;
}

/**
 * Returns the value of the left child.
 * @param index the index of a node in this heap
 * @return the value of the left child of the given node
 */
private Comparable getLeftChild(int index) {
    return elements.get(2 * index);
}

/**
 * Returns the value of the right child.
 * @param index the index of a node in this heap
 * @return the value of the right child of the given node
 */
private Comparable getRightChild(int index) {
    return elements.get(2 * index + 1);
}

/**
 * Returns the value of the parent.
 * @param index the index of a node in this heap
 * @return the value of the parent of the given node
 */
private Comparable getParent(int index) {
    return elements.get(index / 2);
}

private ArrayList<Comparable> elements;
```

16.9 This program demonstrates the use of a heap as a priority queue.

```java
public class HeapDemo {
    /**
     * This program demonstrates the use of a heap as a priority queue.
     */
    public class HeapDemo
```
CHAPTER 16 • Advanced Data Structures

ch16/pqueue/WorkOrder.java

```java
/**
   * This class encapsulates a work order with a priority.
   */
public class WorkOrder implements Comparable
{
    /**
     * Constructs a work order with a given priority and description.
     * @param aPriority the priority of this work order
     * @param aDescription the description of this work order
     */
    public WorkOrder(int aPriority, String aDescription)
    {
        priority = aPriority;
        description = aDescription;
    }

    public String toString()
    {
        return "priority=" + priority + ", description=" + description;
    }

    public int compareTo(Object otherObject)
    {
        WorkOrder other = (WorkOrder) otherObject;
        if (priority < other.priority) return -1;
        if (priority > other.priority) return 1;
        return 0;
    }

    private int priority;
    private String description;
}
```

```java
5 { public static void main(String[] args) {
    MinHeap q = new MinHeap();
    q.add(new WorkOrder(3, "Shampoo carpets"));
    q.add(new WorkOrder(7, "Empty trash"));
    q.add(new WorkOrder(8, "Water plants"));
    q.add(new WorkOrder(10, "Remove pencil sharpener shavings"));
    q.add(new WorkOrder(6, "Replace light bulb"));
    q.add(new WorkOrder(1, "Fix broken sink"));
    q.add(new WorkOrder(9, "Clean coffee maker"));
    q.add(new WorkOrder(2, "Order cleaning supplies"));

    while (q.size() > 0)
    {
        System.out.println(q.remove());
    }
}
```

```java
ch16/pqueue/WorkOrder.java

1 /**
   * This class encapsulates a work order with a priority.
   */
2 public class WorkOrder implements Comparable
3 {
   /**
    * Constructs a work order with a given priority and description.
    * @param aPriority the priority of this work order
    * @param aDescription the description of this work order
    */
4     public WorkOrder(int aPriority, String aDescription)
5     {
         priority = aPriority;
         description = aDescription;
     }

   public String toString()
   {
        return "priority=" + priority + ", description=" + description;
    }

   public int compareTo(Object otherObject)
   {
        WorkOrder other = (WorkOrder) otherObject;
        if (priority < other.priority) return -1;
        if (priority > other.priority) return 1;
        return 0;
    }

    private int priority;
    private String description;
}
```
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 |

**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 $2 \times \text{index}$ and $2 \times \text{index} + 1$?

### 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.

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.

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:

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

```java
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
### HeapSorter.java

```java
/*
 * This class applies the heapsort algorithm to sort an array.
 */
public class HeapSorter {

  /*
   * Constructs a heap sorter that sorts a given array.
   * @param anArray an array of integers
   */
  public HeapSorter(int[] anArray) {
    a = anArray;
  }

  /*
   * Sorts the array managed by this heap sorter.
   */
  public void sort() {
    int n = a.length - 1;
    for (int i = (n - 1) / 2; i >= 0; i--) {
      fixHeap(i, n);
    }
    while (n > 0) {
      swap(0, n);
      n--;
      fixHeap(0, n);
    }
  }

  /*
   * Ensures the heap property for a subtree, provided its
   * children already fulfill the heap property.
   * @param rootIndex the index of the subtree to be fixed
   * @param lastIndex the last valid index of the tree that
   * contains the subtree to be fixed
   */
  private void fixHeap(int rootIndex, int lastIndex) {
    // Remove root
    int rootValue = a[rootIndex];
    // Promote children while they are larger than the root
    int index = rootIndex;
    boolean more = true;
    while (more) {
      int childIndex = getLeftChildIndex(index);
      if (childIndex <= lastIndex) {
        // Use right child instead if it is larger
        int rightChildIndex = getRightChildIndex(index);
        // Other code...
      }
      more = false;
    }
  }
}
```
if (rightChildIndex <= lastIndex && a[rightChildIndex] > a[childIndex])
    childIndex = rightChildIndex;
if (a[childIndex] > rootValue)
    // Promote child
    a[index] = a[childIndex];
    index = childIndex;
else
    // Root value is larger than both children
    more = false;
else
    // No children
    more = false;

// Store root value in vacant slot
a[index] = rootValue;

/**
 Swaps two entries of the array.
 @param i the first position to swap
 @param j the second position to swap
 */
private void swap(int i, int j)
{
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}

/**
 Returns the index of the left child.
 @param index the index of a node in this heap
 @return the index of the left child of the given node
 */
private static int getLeftChildIndex(int index)
{
    return 2 * index + 1;
}
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.
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.
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.)
Review Exercises

★ 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 \(\text{BinarySearchTree.print}()\) 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 \(\text{BinarySearchTree.print}()\) 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?
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:

\[
\text{Carl: B+} \\
\text{Joe: C} \\
\text{Sarah: A}
\]

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.
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

```java
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

```java
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.
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

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

Supply methods

```java
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.
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 \(3 + 4 \times 5\).
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.