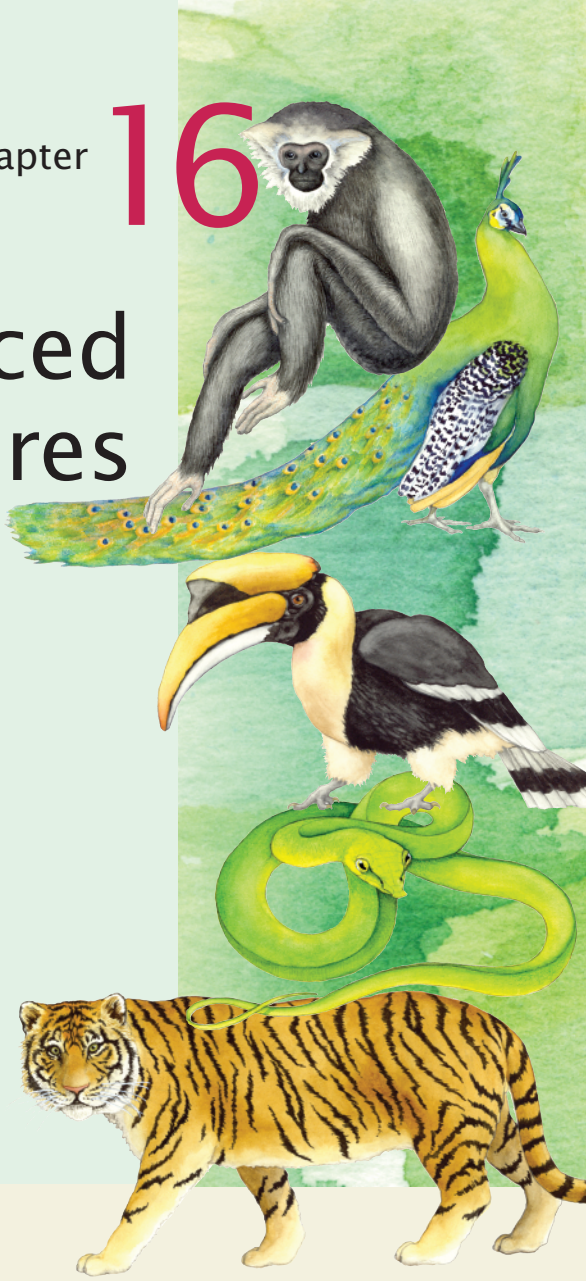
Chapter 16

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 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.

CHAPTER CONTENTS

16.1 Sets

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

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

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:

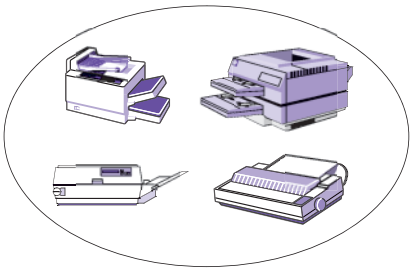
- Adding an element
- Removing an element
- Locating an element (Does the set contain a given object?)
- Listing all elements (not necessarily in the order in which they were added)

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

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 or array list to implement a set. But adding, removing, and containment testing would be $O(n)$ operations, 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.

Figure 1
A Set of Printers



The HashSet and TreeSet classes both implement the Set interface.

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).

When you want to use a set in your program, you must choose between these implementations. In order to use a `HashSet`, the elements must provide a `hashCode` method. We discuss this method in Sections 16.3 and 16.4. Many classes in the standard library implement these methods, for example `String`, `Integer`, `Point`, `Rectangle`, `Color`, and all the collection classes. Therefore, you can form a `HashSet<String>`, `HashSet<Rectangle>`, or even a `HashSet<HashSet<Integer>>`.

The `TreeSet` class uses a different strategy for arranging its elements. Elements are kept in sorted order. In order to use a `TreeSet`, the element type should implement the `Comparable` interface (see Section 15.8). The `String` and `Integer` classes fulfill this requirement, but many other classes do not. You can also construct a `TreeSet` with a `Comparator` (see Special Topic 15.5).

As a rule of thumb, use a hash set unless you want to visit the set elements in sorted order.

Now let's look at using a set of strings. First, construct the set, either as

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

or

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

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

Adding and removing set elements is straightforward:

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

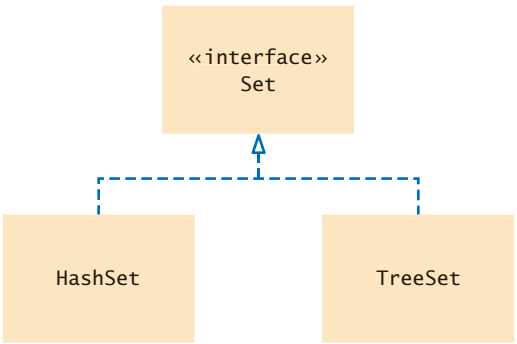


Figure 2
Set Classes and Interfaces in the Standard Library

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

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

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

To visit all elements in a set, use an iterator.

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

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

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

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

A set iterator visits elements in seemingly random order (`HashSet`) or sorted order (`TreeSet`).

Note that the elements are *not* visited in the order in which you inserted them. When you use a hash set, the elements are visited in a seemingly random order—see Section 16.5 for the reason. With a tree set, elements are visited in sorted order.

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

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 shows a practical application of sets. We read in all words from a dictionary file that contains correctly spelled words and place them into a set. We then read all words from a document into a second set—here, the book "Alice in Wonderland". Finally, we print all words from that set that are not in the dictionary set. These are the potential misspellings. (As you can see from the output, we used an American dictionary, and words with British spelling, such as *clamour*, are flagged as potential errors.)

ch16/spellcheck/SpellCheck.java

```
1  import java.util.HashSet;
2  import java.util.Scanner;
3  import java.util.Set;
4  import java.io.File;
5  import java.io.FileNotFoundException;
6
7  /**
8     This program checks which words in a file are not present in a dictionary.
9  */
10 public class SpellCheck
11 {
12    public static void main(String[] args)
13       throws FileNotFoundException
```

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

```
14    {
15       // Read the dictionary and the document
16
17       Set<String> dictionaryWords = readWords("words");
18       Set<String> documentWords = readWords("alice30.txt");
19
20       // Print all words that are in the document but not the dictionary
21
22       for (String word : documentWords)
23       {
24          if (!dictionaryWords.contains(word))
25          {
26             System.out.println(word);
27          }
28       }
29    }
30
31    /**
32       Reads all words from a file.
33       @param filename the name of the file
34       @return a set with all lowercased words in the file. Here, a
35       word is a sequence of upper- and lowercase letters.
36    */
37    public static Set<String> readWords(String filename)
38       throws FileNotFoundException
39    {
40       Set<String> words = new HashSet<String>();
41       Scanner in = new Scanner(new File(filename));
42       // Use any characters other than a-z or A-Z as delimiters
43       in.useDelimiter("[^a-zA-Z]+");
44       while (in.hasNext())
45       {
46          words.add(in.next().toLowerCase());
47       }
48       return words;
49    }
50 }
```

Program Run

```
neighbouring
croqueted
pennyworth
dutchess
comfits
xii
dinn
clamour
. . .
```

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?
3. Suppose you changed line 18 of the `SpellCheck` program to use a `TreeSet` instead of a `HashSet`. How would the output change?
4. When would you choose a tree set over a hash set?

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

Quality Tip 16.1

Use Interface References to Manipulate Data Structures

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

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

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

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

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

Then the method can be used for all set implementations.

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

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

16.2 Maps

A map keeps associations between key and value objects.

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

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.

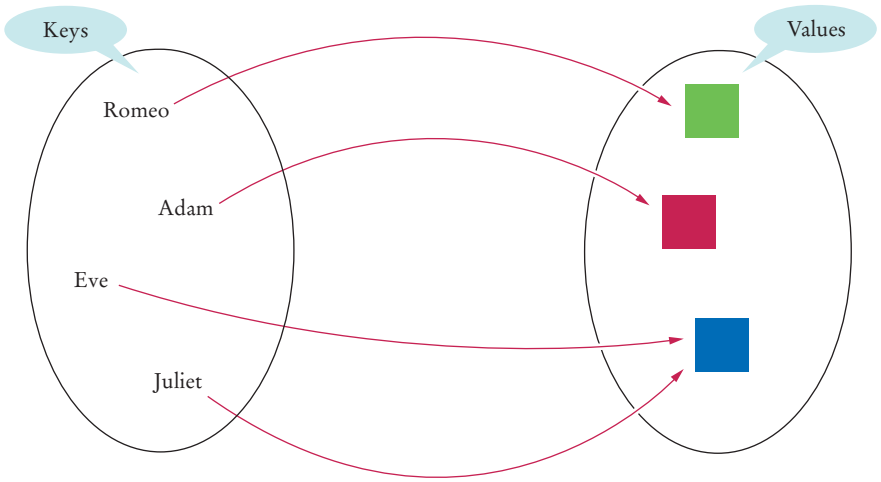


Figure 3 A Map

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

The HashMap and TreeMap classes both implement the Map interface.

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). As with sets, you need to decide which of the two to use. As a rule of thumb, use a hash map unless you want to visit the keys in sorted order.

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

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

or

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

Use the `put` method to add an association:

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

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

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

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

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

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

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

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

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

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

```
Set<String> keySet = m.keySet();
for (String key : keySet)
{
   Color value = m.get(key);
   System.out.println(key + " : " + value);
}
```

When you use a hash map, the keys are visited in a seemingly random order. With a tree map, keys are visited in sorted order. The following sample program shows a map in action.

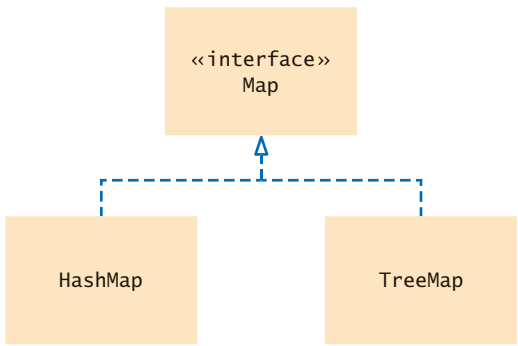


Figure 4
Map Classes and Interfaces in the Standard Library

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

ch16/map/MapDemo.java

```
1  import java.awt.Color;
2  import java.util.HashMap;
3  import java.util.Map;
4  import java.util.Set;
5
6  /**
7     This program demonstrates a map that maps names to colors.
8  */
9  public class MapDemo
10 {
11    public static void main(String[] args)
12    {
13       Map<String, Color> favoriteColors = new HashMap<String, Color>();
14       favoriteColors.put("Juliet", Color.BLUE);
15       favoriteColors.put("Romeo", Color.GREEN);
16       favoriteColors.put("Adam", Color.RED);
17       favoriteColors.put("Eve", Color.BLUE);
18
19       // Print all keys and values in the map
20
21       Set<String> keySet = favoriteColors.keySet();
22       for (String key : keySet)
23       {
24          Color value = favoriteColors.get(key);
25          System.out.println(key + " : " + value);
26       }
27    }
28 }
```

Program Run

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

5. What is the difference between a set and a map?
6. Why is the collection of the keys of a map a set?

Special Topic 16.1

Enhancements to Collection Classes in Java 7

Java 7 provides several syntactical conveniences for working with collection classes.

Type parameters in constructors can be inferred from variable types. You no longer have to repeat them in the variable declaration and the constructor. For example,

```
Set<String> names = new HashSet<>(); // Constructs a HashSet<String>
Map<String, Integer> scores = new TreeMap<>(); // Constructs a TreeMap<String, Integer>
```

You can obtain *collection literals* of type `List`, `Set`, and `Map`, with the following syntax:

```
["Tom", "Diana", "Harry"];
{ 2, 3, 5, 7, 11 };
{ "Juliet" : Color.BLUE, "Romeo" : Color.GREEN, "Eve" : Color.BLUE };
```

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

These objects are immutable: you cannot change the contents of the list, set, or map literal. The objects are instances of classes that implements the `List`, `Set`, and `Map` interfaces, but you don't know what those classes are.

You can pass collection or map literals to methods, for example

```
names.addAll(["Tom", "Diana", "Harry"]);
```

If you want to store a literal in a variable, you must use the interface type for the variable declaration:

```
List<String> friends = ["Tom", "Diana", "Harry"];
```

Alternatively, you can initialize a collection with a literal:

```
ArrayList<String> friends = new ArrayList<>(["Tom", "Diana", "Harry"]);
```

This works because all Java collection and map classes have constructors that copy entries from another collection or map.

Finally, you can use the `[]` operator instead of the `get`, `set`, or `put` methods. For example,

```
String name = names[0];
names[0] = "Fred";
scores["Fred"] = 13;
int score = scores["Fred"];
```

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 values.

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

- Values are accessed by an integer position. Use an `ArrayList`. Go to Step 2, then stop.
- Values are accessed by a key that is not a part of the object. Use a map.
- It doesn't matter. Values are always accessed "in bulk", by traversing the collection and doing something with each value.

Step 2 Determine the element types of key/value types.

For a list or set, determine the type of the elements that you want to store. For example, if you collect a set of books, then the element type is `Book`.

Similarly, for a map, determine the types of the keys and the associated values. If you want to look up books by ID, you can use a `Map<Integer, Book>` or `Map<String, Book>`, depending on your ID type.

Step 3 Determine whether element or key order matters.

When you visit elements from a container or keys from a map, do you care about the order in which they are visited? You have several choices.

- Elements or keys must be sorted. Use a `TreeSet` or `TreeMap`. Go to Step 6.
- Elements must be in the same order in which they were inserted. Your choice is now narrowed down to a `LinkedList` or `ArrayList`.
- It doesn't matter. As long as you get to visit all elements, you don't care in which order. If you chose a map in Step 1, use a `HashMap` and go to Step 5.

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

Step 4 For a collection, determine which operations must be fast.

You have several choices.

- Finding elements must be fast. Use a `HashSet` and go to Step 5.
- Adding and removing elements at the beginning or middle must be fast. Use a `LinkedList`.
- It doesn't matter. You only insert at the end, or collect so few elements that you aren't concerned about speed. Then use an `ArrayList`.

Step 5 For hash sets and maps, decide whether you need to implement the `equals` and `hashCode` methods.

If your elements or keys belong to a class that someone else provided, check whether the class implements `hashCode` and `equals` methods. If so, you are all set. This is the case for most classes in the standard Java library, such as `String`, `Integer`, `Rectangle`, and so on.

If not, decide whether you can compare the elements by identity. Are all elements distinct in your program? That is, can it never happen that you have two different elements with the same instance variables? In that case, you need not do anything—the `hashCode` and `equals` methods of the `Object` class are appropriate.

If you need to implement your own `equals` and `hashCode` methods, turn to Section 16.4.

Step 6 If you use a tree, decide whether to supply a comparator.

Look at the class of the set elements or map keys. 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. This is the case for many classes in the standard library, in particular for `String` and `Integer`.

If no, then your element class must implement the `Comparable` interface, or you must provide a class that implements the `Comparator` interface. See Section 14.8 for the details.

Worked Example 16.1

Word Frequency

In this Worked Example, we read a text file and print a list of all words in the file in alphabetical order, together with a count that indicates how often each word occurred in the file

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 computes an integer value from an object.

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 override. The call

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

computes the hash code of the object `x`.

Available online in WileyPLUS and at `www.wiley.com/college/horstmann`.

Table 1 Sample Strings and Their Hash Codes

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

Table 1 shows some examples of strings and their hash codes. You will see in Section 16.4 how these values are obtained.

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

It is possible for two or more distinct objects to have the same hash code; this is called a *collision*. For example, the strings `"VII"` and `"Ugh"` happen to have the same hash code. These collisions are very rare for strings (see Exercise P16.6).

Section 16.5 explains how you should override 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).

If there are no collisions, 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!

Of course, it is not feasible to allocate an array that is large enough to hold all possible integer index positions. Therefore, we must pick an array of some reasonable size and then reduce the hash code to fall inside the array:

```
int h = x.hashCode();
if (h < 0) h = -h;
position = h % buckets.length;
```

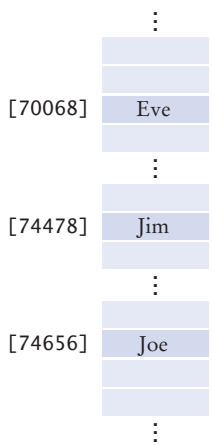
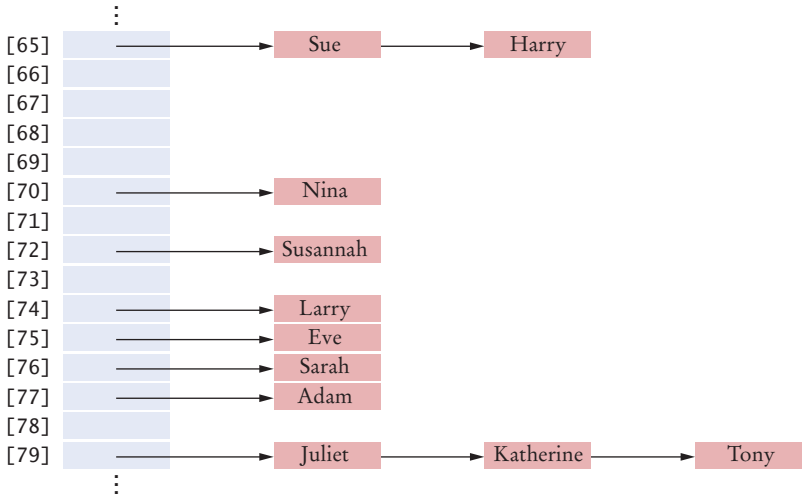


Figure 5
A Simplistic Implementation of a Hash Table

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

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



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

After reducing the hash code modulo a smaller array size, it becomes even more likely that several objects will collide. In order to handle collisions, we will store all colliding elements in a "bucket", a linked list of elements with the same position value (see Figure 6).

Here is the algorithm for finding an object `x` in a hash table.

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.

Adding an element is a straightforward 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.

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

In the best case, in which there are no collisions, all buckets either are empty or have a single element. Then adding, finding, and removing elements takes constant or $O(1)$ time.

More generally, for this algorithm to be effective, the bucket sizes must be small. (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 chances 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,

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

the hash table size should be chosen to be a prime number to minimize the number of collisions.

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

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

ch16/hashtable/HashSet.java

```
1  import java.util.AbstractSet;
2  import java.util.Iterator;
3  import java.util.NoSuchElementException;
4
5  /**
6     A hash set stores an unordered collection of objects, using
7     a hash table.
8  */
9  public class HashSet extends AbstractSet
10 {
11    private Node[] buckets;
12    private int size;
13
14    /**
15       Constructs a hash table.
16       @param bucketsLength the length of the buckets array
17    */
18    public HashSet(int bucketsLength)
19    {
20       buckets = new Node[bucketsLength];
21       size = 0;
22    }
23
24    /**
25       Tests for set membership.
26       @param x an object
27       @return true if x is an element of this set
28    */
29    public boolean contains(Object x)
30    {
31       int h = x.hashCode();
32       if (h < 0) h = -h;
33       h = h % buckets.length;
34
35       Node current = buckets[h];
36       while (current != null)
37       {
38          if (current.data.equals(x)) return true;
39          current = current.next;
40       }
41       return false;
42    }
43
```

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

```
44    /**
45       Adds an element to this set.
46       @param x an object
47       @return true if x is a new object, false if x was
48       already in the set
49    */
50    public boolean add(Object x)
51    {
52       int h = x.hashCode();
53       if (h < 0) h = -h;
54       h = h % buckets.length;
55
56       Node current = buckets[h];
57       while (current != null)
58       {
59          if (current.data.equals(x))
60             return false; // Already in the set
61          current = current.next;
62       }
63       Node newNode = new Node();
64       newNode.data = x;
65       newNode.next = buckets[h];
66       buckets[h] = newNode;
67       size++;
68       return true;
69    }
70
71    /**
72       Removes an object from this set.
73       @param x an object
74       @return true if x was removed from this set, false
75       if x was not an element of this set
76    */
77    public boolean remove(Object x)
78    {
79       int h = x.hashCode();
80       if (h < 0) h = -h;
81       h = h % buckets.length;
82
83       Node current = buckets[h];
84       Node previous = null;
85       while (current != null)
86       {
87          if (current.data.equals(x))
88          {
89             if (previous == null) buckets[h] = current.next;
90             else previous.next = current.next;
91             size--;
92             return true;
93          }
94          previous = current;
95          current = current.next;
96       }
97       return false;
98    }
99
```

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

```
100    /**
101       Returns an iterator that traverses the elements of this set.
102       @return a hash set iterator
103    */
104    public Iterator iterator()
105    {
106       return new HashSetIterator();
107    }
108
109    /**
110       Gets the number of elements in this set.
111       @return the number of elements
112    */
113    public int size()
114    {
115       return size;
116    }
117
118    class Node
119    {
120       public Object data;
121       public Node next;
122    }
123
124    class HashSetIterator implements Iterator
125    {
126       private int bucket;
127       private Node current;
128       private int previousBucket;
129       private Node previous;
130
131       /**
132          Constructs a hash set iterator that points to the
133          first element of the hash set.
134       */
135       public HashSetIterator()
136       {
137          current = null;
138          bucket = -1;
139          previous = null;
140          previousBucket = -1;
141       }
142
143       public boolean hasNext()
144       {
145          if (current != null && current.next != null)
146             return true;
147          for (int b = bucket + 1; b < buckets.length; b++)
148             if (buckets[b] != null) return true;
149          return false;
150       }
151
152       public Object next()
153       {
154          previous = current;
155          previousBucket = bucket;
156
```

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

```
157          if (current == null || current.next == null)
158          {
159             // Move to next bucket
160             bucket++;
161
162             while (bucket < buckets.length
163                   && buckets[bucket] == null)
164                bucket++;
165             if (bucket < buckets.length)
166                current = buckets[bucket];
167             else
168                throw new NoSuchElementException();
169          }
170          else // Move to next element in bucket
171             current = current.next;
172          return current.data;
173       }
174
175       public void remove()
176       {
177          if (previous != null && previous.next == current)
178             previous.next = current.next;
179          else if (previousBucket < bucket)
180             buckets[bucket] = current.next;
181          else
182             throw new IllegalStateException();
183          current = previous;
184          bucket = previousBucket;
185       }
186    }
187 }
```

ch16/hashtable/HashSetDemo.java

```
1  import java.util.Iterator;
2  import java.util.Set;
3
4  /**
5     This program demonstrates the hash set class.
6  */
7  public class HashSetDemo
8  {
9     public static void main(String[] args)
10    {
11       Set names = new HashSet(101); // 101 is a prime
12
13       names.add("Harry");
14       names.add("Sue");
15       names.add("Nina");
16       names.add("Susannah");
17       names.add("Larry");
18       names.add("Eve");
19       names.add("Sarah");
20       names.add("Adam");
21       names.add("Tony");
22       names.add("Katherine");
23       names.add("Juliet");
```

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

```
24          names.add("Romeo");
25          names.remove("Romeo");
26          names.remove("George");
27
28          Iterator iter = names.iterator();
29          while (iter.hasNext())
30             System.out.println(iter.next());
31       }
32    }
```

Program Run

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

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

16.4 Computing Hash Codes

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

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

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

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

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

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

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 A to look up the character values: 'a' is 97, 'e' is 101, and 't' is 116.)

When implementing the hashCode method, combine the hash codes for the instance variables.

For your own classes, you should make up a hash code that combines the hash codes of the instance variables in a similar way. For example, let us implement a hashCode method for the Coin class. There are two instance variables: 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.

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

Then combine the two hash codes.

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

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

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

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

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

Your hashCode method must be compatible with the equals method.

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

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

will also be equal, because the hash code is computed from the hash codes of the `name` and `value` instance variables.

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

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

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

If a class provides neither `equals` nor `hashCode`, then objects are compared by identity.

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

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 provided or it has not been provided. If `equals` has not been provided, only identical objects are considered equal. In that case, don't provide `hashCode` either. However, if the `equals` method has been provided, look at its implementation. Typically, two objects are considered equal if some or all of the instance variables are equal. Sometimes, not all instance variables are used in the comparison. Two `Student` objects may be considered equal if their `studentID` variables are equal. Implement the `hashCode` method to combine the hash codes of the instance variables that are compared in the `equals` method.

In a hash map, only the keys are hashed.

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

ch16/hashcode/Coin.java

```
1  /**
2     A coin with a monetary value.
3  */
4  public class Coin
5  {
6     private double value;
7     private String name;
8
9     /**
10       Constructs a coin.
11       @param aValue the monetary value of the coin
12       @param aName the name of the coin
13    */
14    public Coin(double aValue, String aName)
15    {
```

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

```
16         value = aValue;
17         name = aName;
18      }
19
20      /**
21         Gets the coin value.
22         @return the value
23      */
24      public double getValue()
25      {
26         return value;
27      }
28
29      /**
30         Gets the coin name.
31         @return the name
32      */
33      public String getName()
34      {
35         return name;
36      }
37
38      public boolean equals(Object otherObject)
39      {
40         if (otherObject == null) return false;
41         if (getClass() != otherObject.getClass()) return false;
42         Coin other = (Coin) otherObject;
43         return value == other.value && name.equals(other.name);
44      }
45
46      public int hashCode()
47      {
48         int h1 = name.hashCode();
49         int h2 = new Double(value).hashCode();
50         final int HASH_MULTIPLIER = 29;
51         int h = HASH_MULTIPLIER * h1 + h2;
52         return h;
53      }
54
55      public String toString()
56      {
57         return "Coin[value=" + value + ",name=" + name + "]";
58      }
59   }
```

ch16/hashcode/CoinHashCodePrinter.java

```
1  import java.util.HashSet;
2  import java.util.Set;
3
4  /**
5     A program that prints hash codes of coins.
6  */
7  public class CoinHashCodePrinter
8  {
9     public static void main(String[] args)
10    {
```

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

```
11       Coin coin1 = new Coin(0.25, "quarter");
12       Coin coin2 = new Coin(0.25, "quarter");
13       Coin coin3 = new Coin(0.05, "nickel");
14
15       System.out.println("hash code of coin1=" + coin1.hashCode());
16       System.out.println("hash code of coin2=" + coin2.hashCode());
17       System.out.println("hash code of coin3=" + coin3.hashCode());
18
19       Set<Coin> coins = new HashSet<Coin>();
20       coins.add(coin1);
21       coins.add(coin2);
22       coins.add(coin3);
23
24       for (Coin c : coins)
25          System.out.println(c);
26    }
27 }
```

Program Run

```
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]
```

9. What is the hash code of the string `"to"`?
10. What is the hash code of `new Integer(13)`?

Common Error 16.1

Forgetting to Provide `hashCode`

When putting elements into a hash table, make sure that the `hashCode` method is provided. (The only exception is that you don't need to provide `hashCode` if `equals` isn't provided either. 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* provide the `hashCode` method of the `Coin` class. Then the following code is likely to fail:

```
Set<Coin> coins = new HashSet<Coin>();
coins.add(new Coin(0.25, "quarter"));
// The following comparison will probably fail if hashCode not provided
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 provide a `hashCode` method in the `Coin` class.

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

16.5 Binary Search Trees

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

If we use an array to store the elements of a set, inserting or removing an element is an $O(n)$ operation. In this section, you will see how tree-shaped data structures can keep elements in sorted order with more efficient insertion and removal.

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

A linked list is a one-dimensional data structure. In a linked list, a node has only one successor. 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 keeping with the tree image, the node at the top is called the *root node*, and the nodes without children are called *leaf nodes*. 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 constructed to have this 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.

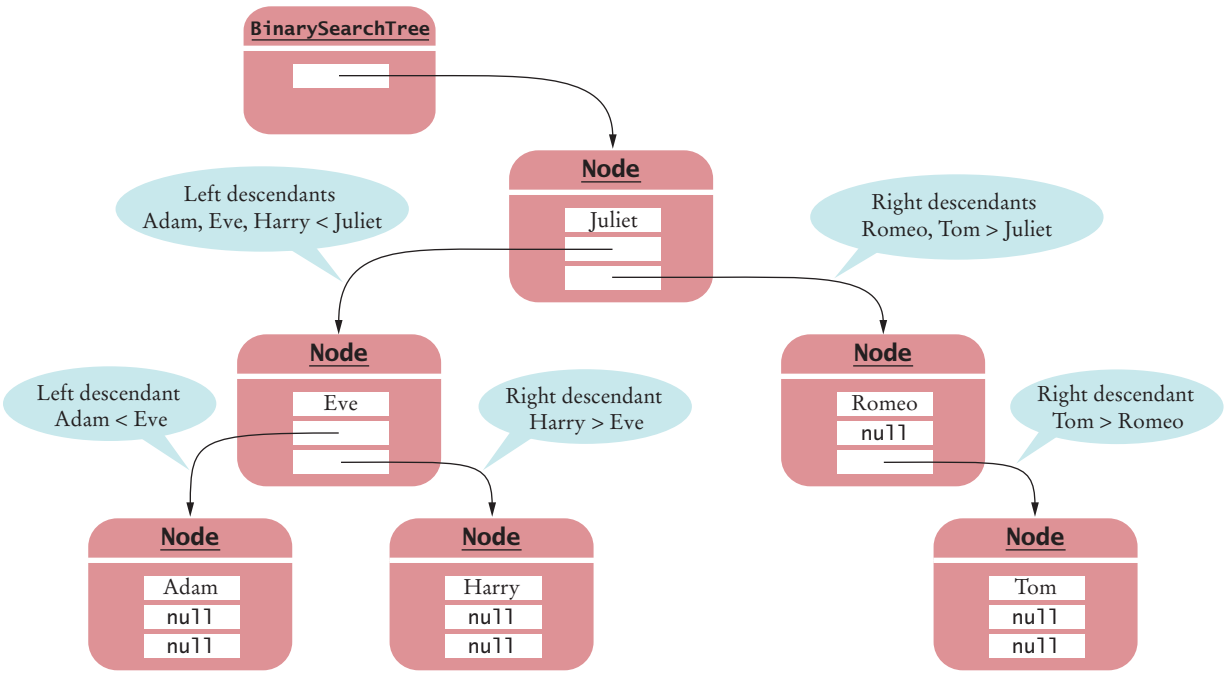


Figure 7 A Binary Search Tree

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

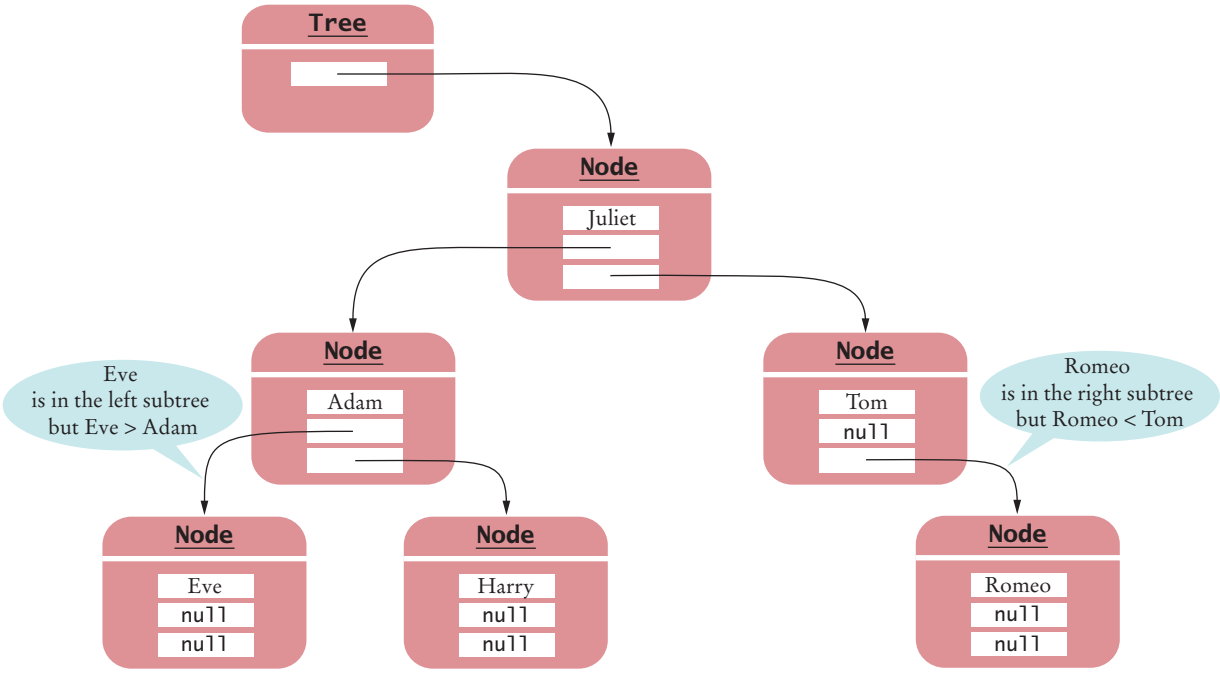


Figure 8 A Binary Tree That Is Not a Binary Search Tree

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.

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

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 an instance variable `data`. At the fringes of the tree, one or two of the child references can be `null`. The `data` variable 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.

```
public class BinarySearchTree
{
   private Node root;

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

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

```
        public void addNode(Node newNode) { . . . }
        . . .
    }
}
```

To insert a value into a binary search tree, keep comparing the value with the node data and follow the nodes to the left or right, until reaching a `null` node.

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.

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

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

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

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

Start with the root node, `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).

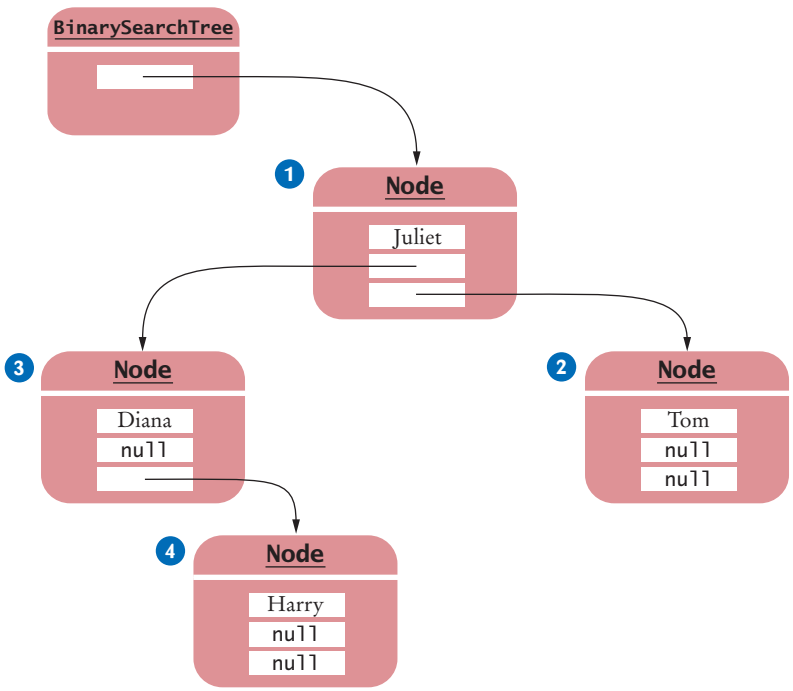


Figure 9
Binary Search Tree After Four Insertions

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

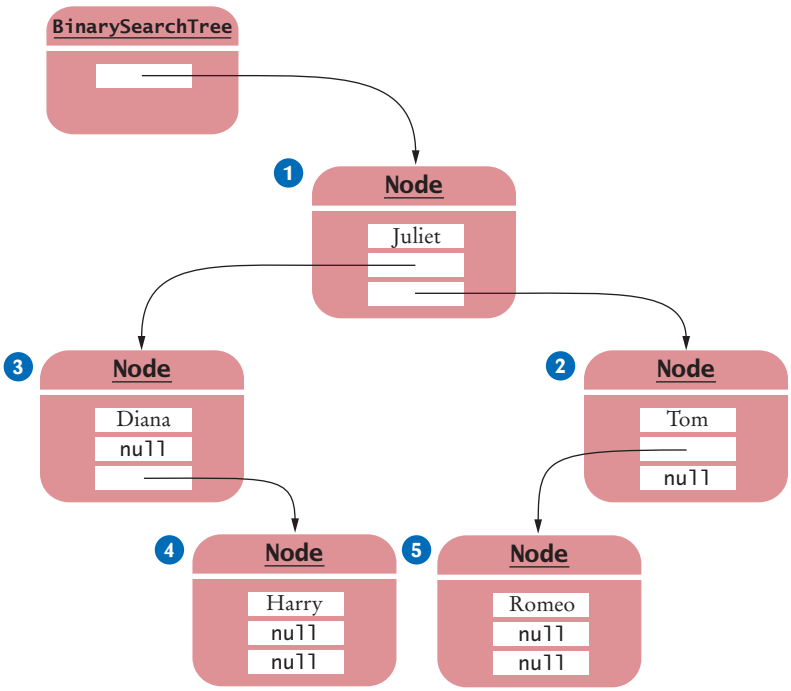


Figure 10
Binary Search Tree
After Five Insertions

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:

```
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:

```
class Node
{
   . . .
   public void addNode(Node newNode)
   {
```

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

```
            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's 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).

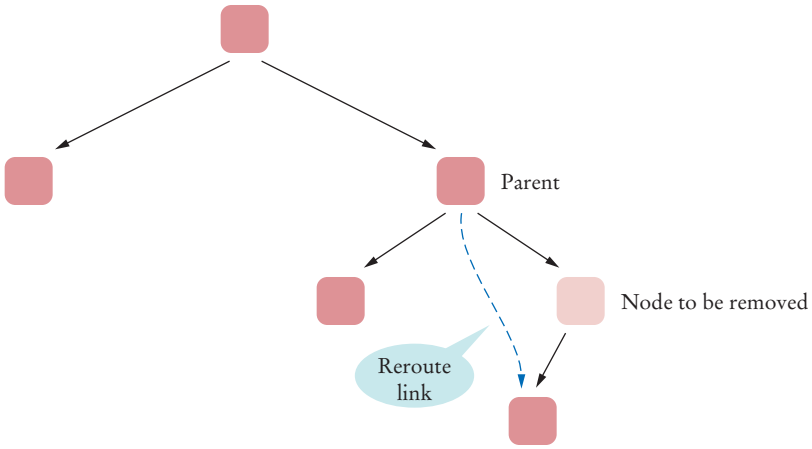


Figure 11
Removing a Node with One Child

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

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

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

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

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

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 P16.21).

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.

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.

In a balanced tree, all paths from the root to the leaves have about the same length.

Now that you have seen the implementation of this 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 or removed; that is an advantage. How fast insertion and removal are, however, depends on the shape of the tree. These operations are fast if the tree is *balanced* (see Figure 13).

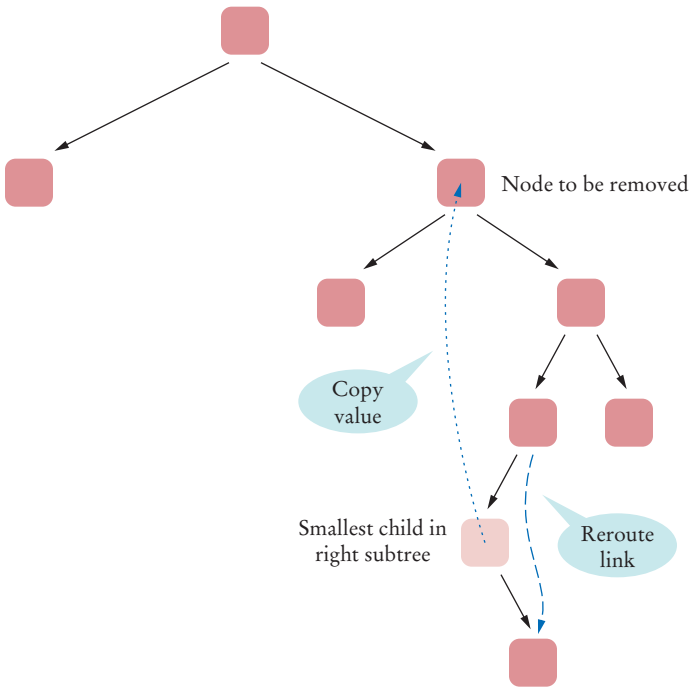


Figure 12 Removing a Node with Two Children

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

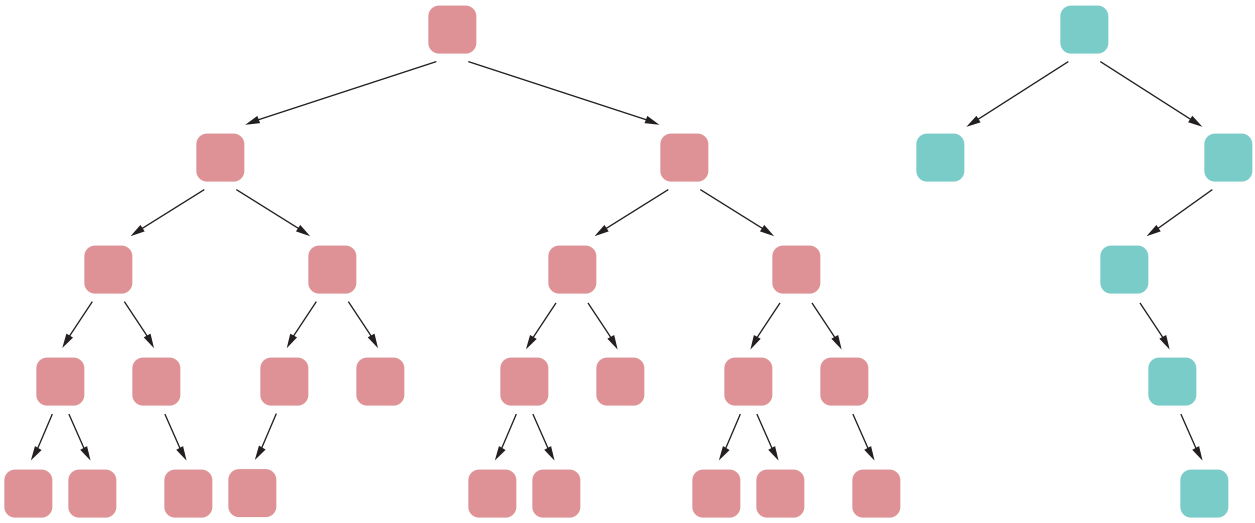


Figure 13 Balanced and Unbalanced Trees

Adding, finding, and removing an element in a tree set is proportional to the height of the tree.

In a balanced tree, all paths from the root to one of the leaf nodes (that is, nodes without children) have approximately the same length. The number of nodes in the longest of these paths is called the *height* of the tree. The trees in Figure 13 have height 5.

Because the operations of finding, adding, and removing an element process the nodes along a path from the root to a leaf, their execution time is proportional to the height of the tree, and not to the total number of nodes in the tree.

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

A tree of height h can have up to $n = 2^h - 1$ nodes. For example, a completely filled tree of height 4 has $1 + 2 + 4 + 8 = 15 = 2^4 - 1$ nodes. In other words, $h = \log_2(n + 1)$ for a completely filled tree. For a balanced tree, we still have $h \approx \log_2 n$. For example, the height of a tree with 1,000 nodes is approximately 10 (because $1024 = 2^{10}$). A tree with 1,000,000 nodes has height approximately 20. In such a tree, you can find any element in about 20 steps. That is a lot faster than traversing the 1,000,000 elements of a list.

On the other hand, if the tree happens to be *unbalanced*, then binary tree operations can be slow—in the worst case, as slow as insertion into a linked list.

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. The standard Java library uses *red-black trees*, a special form of balanced binary trees, to implement sets and maps.

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

ch16/tree/BinarySearchTree.java

```
1  /**
2     This class implements a binary search tree whose
3     nodes hold objects that implement the Comparable
4     interface.
5  */
6  public class BinarySearchTree
7  {
8     private Node root;
9
10    /**
11       Constructs an empty tree.
12    */
13    public BinarySearchTree()
14    {
15       root = null;
16    }
17
18    /**
19       Inserts a new node into the tree.
20       @param obj the object to insert
21    */
22    public void add(Comparable obj)
23    {
24       Node newNode = new Node();
25       newNode.data = obj;
26       newNode.left = null;
27       newNode.right = null;
28       if (root == null) root = newNode;
29       else root.addNode(newNode);
30    }
31
32    /**
33       Tries to find an object in the tree.
34       @param obj the object to find
35       @return true if the object is contained in the tree
36    */
37    public boolean find(Comparable obj)
38    {
39       Node current = root;
40       while (current != null)
41       {
42          int d = current.data.compareTo(obj);
43          if (d == 0) return true;
44          else if (d > 0) current = current.left;
45          else current = current.right;
46       }
47       return false;
48    }
49
50    /**
51       Tries to remove an object from the tree. Does nothing
52       if the object is not contained in the tree.
53       @param obj the object to remove
54    */
55    public void remove(Comparable obj)
56    {
57       // Find node to be removed
58
```

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

```
59          Node toBeRemoved = root;
60          Node parent = null;
61          boolean found = false;
62          while (!found && toBeRemoved != null)
63          {
64             int d = toBeRemoved.data.compareTo(obj);
65             if (d == 0) found = true;
66             else
67             {
68                parent = toBeRemoved;
69                if (d > 0) toBeRemoved = toBeRemoved.left;
70                else toBeRemoved = toBeRemoved.right;
71             }
72          }
73
74          if (!found) return;
75
76          // toBeRemoved contains obj
77
78          // If one of the children is empty, use the other
79
80          if (toBeRemoved.left == null || toBeRemoved.right == null)
81          {
82             Node newChild;
83             if (toBeRemoved.left == null)
84                newChild = toBeRemoved.right;
85             else
86                newChild = toBeRemoved.left;
87
88             if (parent == null) // Found in root
89                root = newChild;
90             else if (parent.left == toBeRemoved)
91                parent.left = newChild;
92             else
93                parent.right = newChild;
94             return;
95          }
96
97          // Neither subtree is empty
98
99          // Find smallest element of the right subtree
100
101         Node smallestParent = toBeRemoved;
102         Node smallest = toBeRemoved.right;
103         while (smallest.left != null)
104         {
105            smallestParent = smallest;
106            smallest = smallest.left;
107         }
108
109         // smallest contains smallest child in right subtree
110
111         // Move contents, unlink child
112
113         toBeRemoved.data = smallest.data;
114         if (smallestParent == toBeRemoved)
115            smallestParent.right = smallest.right;
116         else
117            smallestParent.left = smallest.right;
```

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

```
118    }
119
120    /**
121       Prints the contents of the tree in sorted order.
122    */
123    public void print()
124    {
125       if (root != null)
126          root.printNodes();
127       System.out.println();
128    }
129
130    /**
131       A node of a tree stores a data item and references
132       to the child nodes to the left and to the right.
133    */
134    class Node
135    {
136       public Comparable data;
137       public Node left;
138       public Node right;
139
140       /**
141          Inserts a new node as a descendant of this node.
142          @param newNode the node to insert
143       */
144       public void addNode(Node newNode)
145       {
146          int comp = newNode.data.compareTo(data);
147          if (comp < 0)
148          {
149             if (left == null) left = newNode;
150             else left.addNode(newNode);
151          }
152          if (comp > 0)
153          {
154             if (right == null) right = newNode;
155             else right.addNode(newNode);
156          }
157       }
158
159       /**
160          Prints this node and all of its descendants
161          in sorted order.
162       */
163       public void printNodes()
164       {
165          if (left != null)
166             left.printNodes();
167          System.out.print(data + " ");
168          if (right != null)
169             right.printNodes();
170       }
171    }
172 }
```

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

11. What is the difference between a tree, a binary tree, and a balanced binary tree?
12. Give an example of a string that, when inserted into the tree of Figure 10, becomes a right child of `Romeo`.

16.6 Binary Tree Traversal

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

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

Let's try this out with the tree in Figure 10 on page 689. The algorithm tells us to

1. Print the left subtree of `Juliet`; that is, `Diana` 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 `Diana`?

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

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

```
Diana 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:

```
Diana Harry Juliet Romeo Tom
```

The tree is printed in sorted order.

Now we can implement the `print` method. You need a worker method `printNodes` of the `Node` class:

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

```
class Node
{
   . . .
   public void printNodes()
   {
      if (left != null)
         left.printNodes();
      System.out.print(data + " ");

      if (right != null)
         right.printNodes();
   }
   . . .
}
```

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

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

To visit all elements in a tree, visit the root and recursively visit the subtrees. We distinguish between preorder, inorder, and postorder traversal.

This visitation scheme is called *inorder traversal* (visit the left subtree, the root, the right subtree). There are two other common 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) * 5
3 + 4 * 5
```

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

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

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

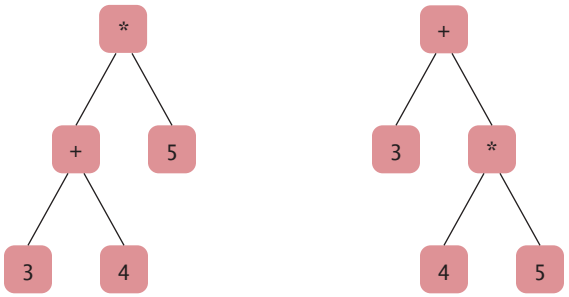


Figure 14 Expression Trees

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

```
3 4 + 5 *
```

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

whereas the second tree yields

```
3 4 5 * +
```

You can interpret these sequences as expressions in "reverse Polish notation" (see Special Topic 15.1), or equivalently, instructions for a stack-based calculator (see Worked Example 15.1).

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

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

16.7 Priority Queues

In Section 15.4, you encountered two common abstract data types: stacks and queues. Another important abstract data type, the **priority queue**, collects elements, each of which has a *priority*. A typical example of a priority queue is a collection of work requests, some of which may be more urgent than others. Unlike a regular queue, the priority queue does not maintain a first-in, first-out discipline. Instead, elements are retrieved according to their priority. In other words, new items can be inserted in any order. But whenever an item is removed, that item has highest priority.

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

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

For example, consider this sample code:

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

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

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

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.

16.8 Heaps

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

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

1. A heap is *almost completely filled:* all nodes are filled in, except the last level may have some nodes missing toward the right (see Figure 15).
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 16).

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

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

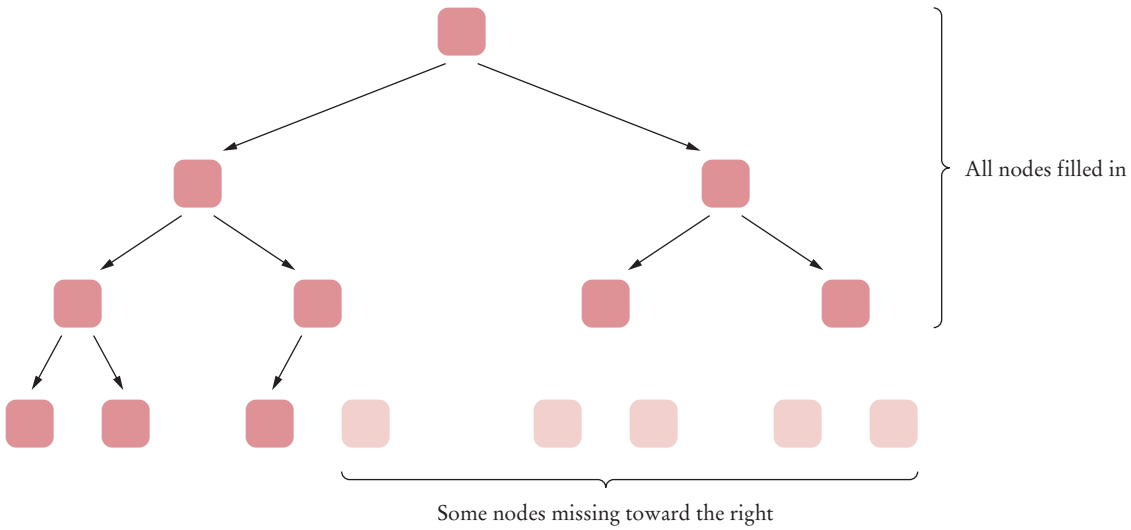
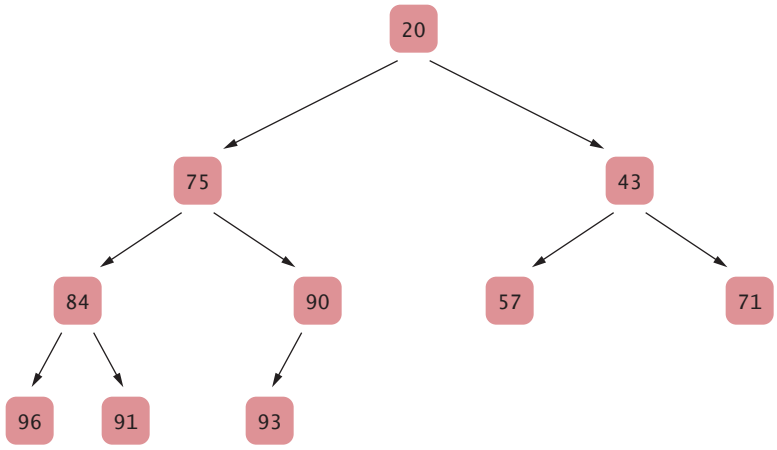


Figure 15 An Almost Completely Filled Tree

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

Figure 16
A Heap



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 you 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 17).

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

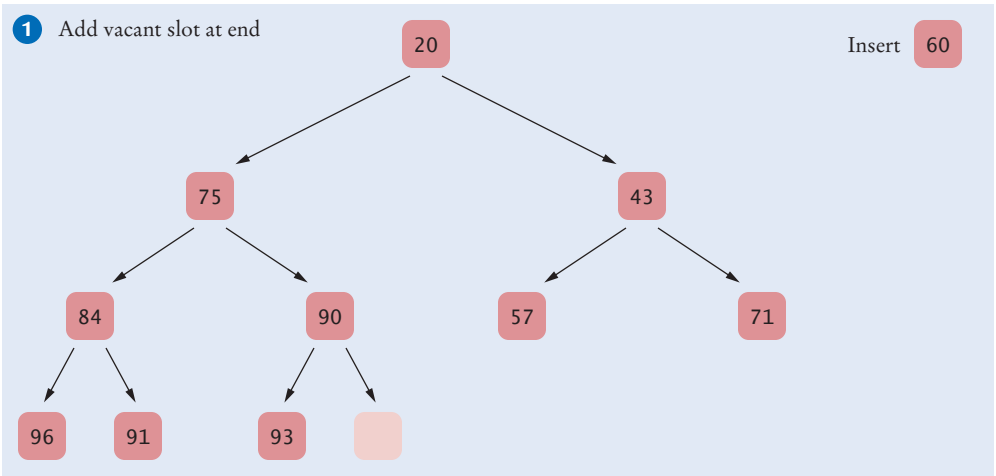


Figure 17 Inserting an Element into a Heap

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

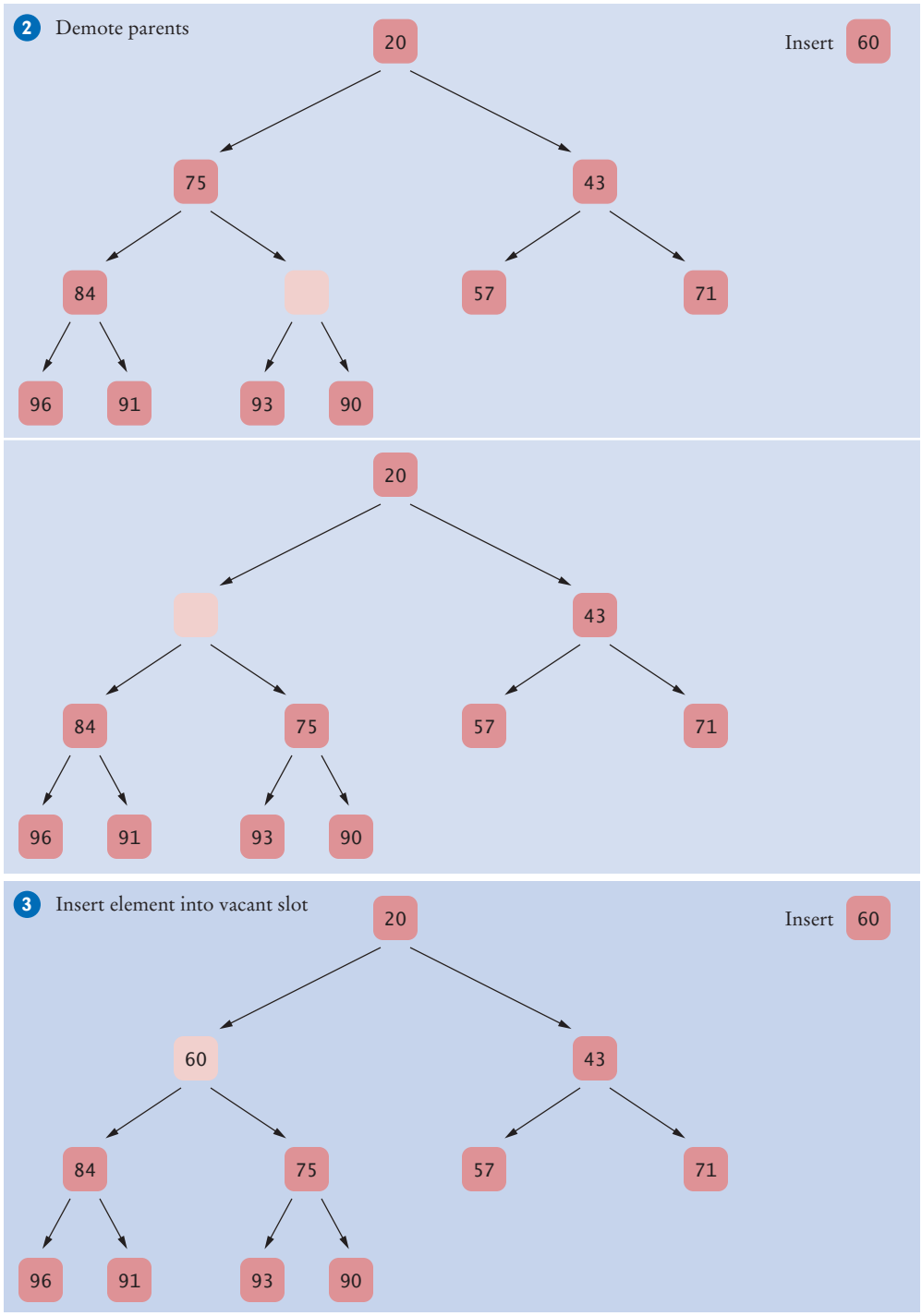


Figure 17 (continued) Inserting an Element into a Heap

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.

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

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 18 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 18 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".

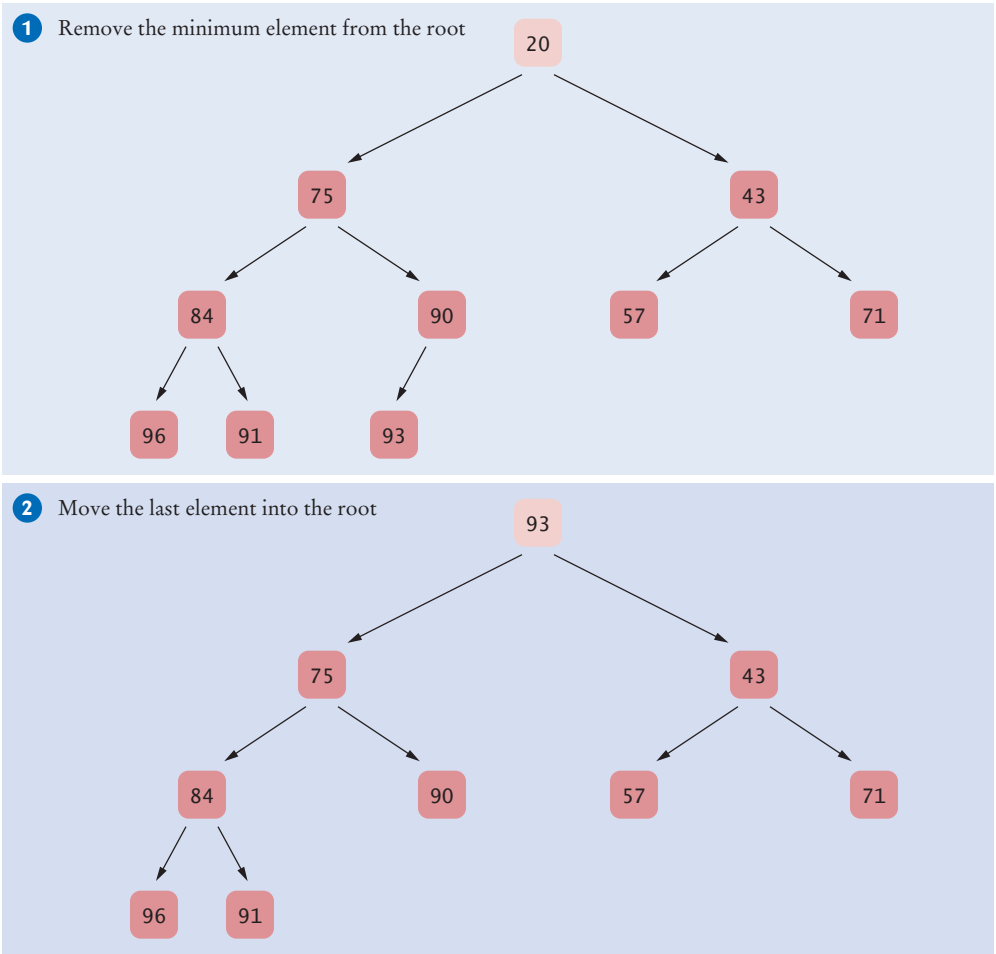


Figure 18 Removing the Minimum Value from a Heap

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

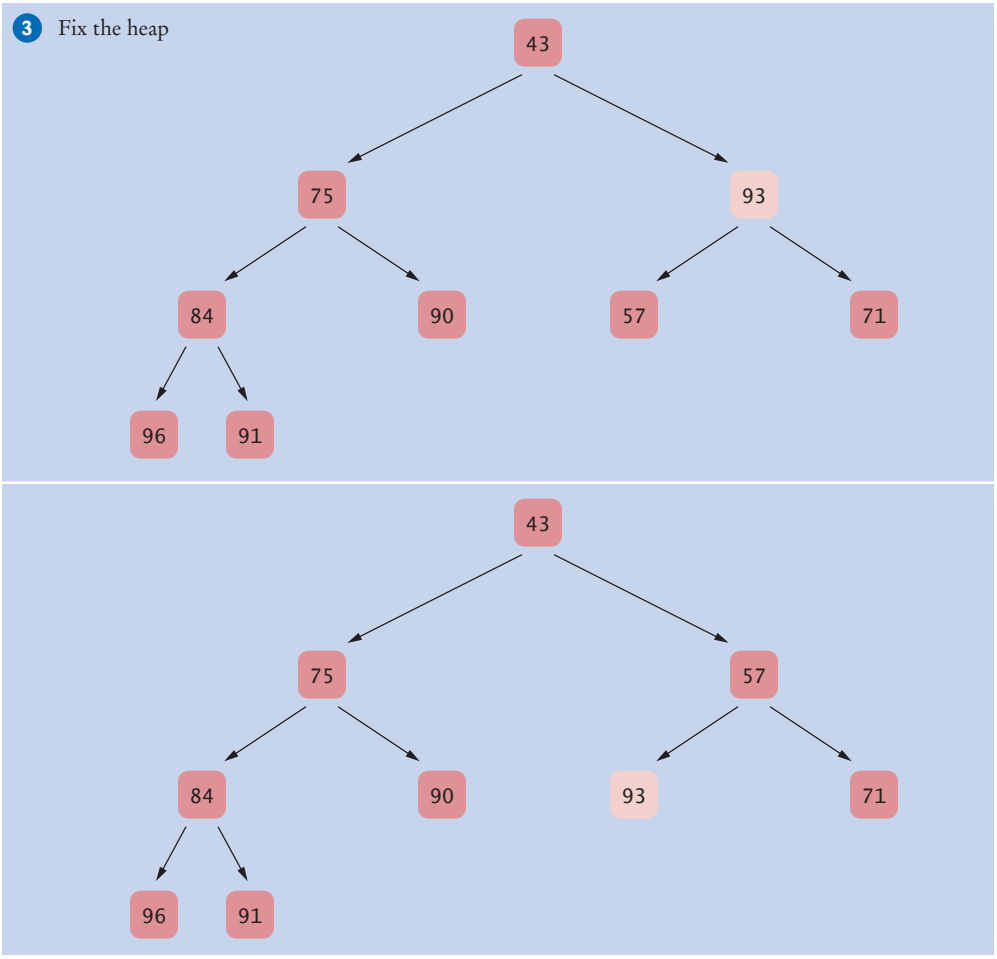


Figure 18 (continued) Removing the Minimum Value from a Heap

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

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

or

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

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

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.

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

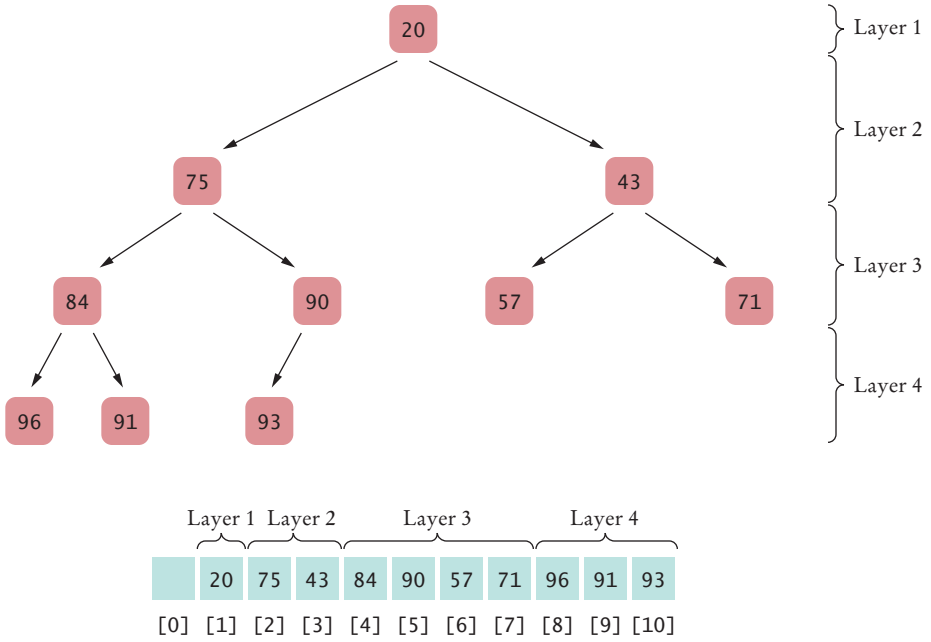


Figure 19 Storing a Heap in an Array

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

Heaps have another major advantage. Because of the regular layout of the heap nodes, it is easy to store the node values in an array or array list. First store the first layer, then the second, and so on (see Figure 19). 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 19, 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.

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.

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

ch16/pqueue/MinHeap.java

```
1  import java.util.*;
2
3  /**
4     This class implements a heap.
5  */
6  public class MinHeap
7  {
8     private ArrayList<Comparable> elements;
9
10    /**
11       Constructs an empty heap.
12    */
13    public MinHeap()
14    {
15       elements = new ArrayList<Comparable>();
16       elements.add(null);
17    }
18
19    /**
20       Adds a new element to this heap.
21       @param newElement the element to add
22    */
23    public void add(Comparable newElement)
24    {
25       // Add a new leaf
26       elements.add(null);
27       int index = elements.size() - 1;
28
29       // Demote parents that are larger than the new element
30       while (index > 1
31             && getParent(index).compareTo(newElement) > 0)
32       {
33          elements.set(index, getParent(index));
34          index = getParentIndex(index);
35       }
36
37       // Store the new element in the vacant slot
38       elements.set(index, newElement);
39    }
40
41    /**
42       Gets the minimum element stored in this heap.
43       @return the minimum element
44    */
45    public Comparable peek()
46    {
47       return elements.get(1);
48    }
49
50    /**
51       Removes the minimum element from this heap.
52       @return the minimum element
53    */
54    public Comparable remove()
55    {
56       Comparable minimum = elements.get(1);
57
```

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

```
58          // Remove last element
59          int lastIndex = elements.size() - 1;
60          Comparable last = elements.remove(lastIndex);
61
62          if (lastIndex > 1)
63          {
64             elements.set(1, last);
65             fixHeap();
66          }
67
68          return minimum;
69       }
70
71       /**
72          Turns the tree back into a heap, provided only the root
73          node violates the heap condition.
74       */
75       private void fixHeap()
76       {
77          Comparable root = elements.get(1);
78
79          int lastIndex = elements.size() - 1;
80          // Promote children of removed root while they are smaller than last
81
82          int index = 1;
83          boolean more = true;
84          while (more)
85          {
86             int childIndex = getLeftChildIndex(index);
87             if (childIndex <= lastIndex)
88             {
89                // Get smaller child
90
91                // Get left child first
92                Comparable child = getLeftChild(index);
93
94                // Use right child instead if it is smaller
95                if (getRightChildIndex(index) <= lastIndex
96                      && getRightChild(index).compareTo(child) < 0)
97                {
98                   childIndex = getRightChildIndex(index);
99                   child = getRightChild(index);
100               }
101
102               // Check if larger child is smaller than root
103               if (child.compareTo(root) < 0)
104               {
105                  // Promote child
106                  elements.set(index, child);
107                  index = childIndex;
108               }
109               else
110               {
111                  // Root is smaller than both children
112                  more = false;
113               }
114            }
115            else
116            {
```

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

```
117             // No children
118             more = false;
119          }
120       }
121
122       // Store root element in vacant slot
123       elements.set(index, root);
124    }
125
126    /**
127       Returns the number of elements in this heap.
128    */
129    public int size()
130    {
131       return elements.size() - 1;
132    }
133
134    /**
135       Returns the index of the left child.
136       @param index the index of a node in this heap
137       @return the index of the left child of the given node
138    */
139    private static int getLeftChildIndex(int index)
140    {
141       return 2 * index;
142    }
143
144    /**
145       Returns the index of the right child.
146       @param index the index of a node in this heap
147       @return the index of the right child of the given node
148    */
149    private static int getRightChildIndex(int index)
150    {
151       return 2 * index + 1;
152    }
153
154    /**
155       Returns the index of the parent.
156       @param index the index of a node in this heap
157       @return the index of the parent of the given node
158    */
159    private static int getParentIndex(int index)
160    {
161       return index / 2;
162    }
163
164    /**
165       Returns the value of the left child.
166       @param index the index of a node in this heap
167       @return the value of the left child of the given node
168    */
169    private Comparable getLeftChild(int index)
170    {
171       return elements.get(2 * index);
172    }
173
```

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

```
174    /**
175       Returns the value of the right child.
176       @param index the index of a node in this heap
177       @return the value of the right child of the given node
178    */
179    private Comparable getRightChild(int index)
180    {
181       return elements.get(2 * index + 1);
182    }
183
184    /**
185       Returns the value of the parent.
186       @param index the index of a node in this heap
187       @return the value of the parent of the given node
188    */
189    private Comparable getParent(int index)
190    {
191       return elements.get(index / 2);
192    }
193 }
```

ch16/pqueue/WorkOrder.java

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

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

ch16/pqueue/HeapDemo.java

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

Program Run

```
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 * index` and `2 * index + 1`?

16.9 The Heapsort Algorithm

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

Heapsort is an $O(n \log(n))$ 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,

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

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-last level of the tree.

The sorting algorithm uses a generalized `fixHeap` method that fixes a subtree:

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

The subtree is specified by the index of its root and of its last node.

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 20).

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

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

It can be shown that this procedure turns an arbitrary array into a heap in $O(n)$ steps.

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.

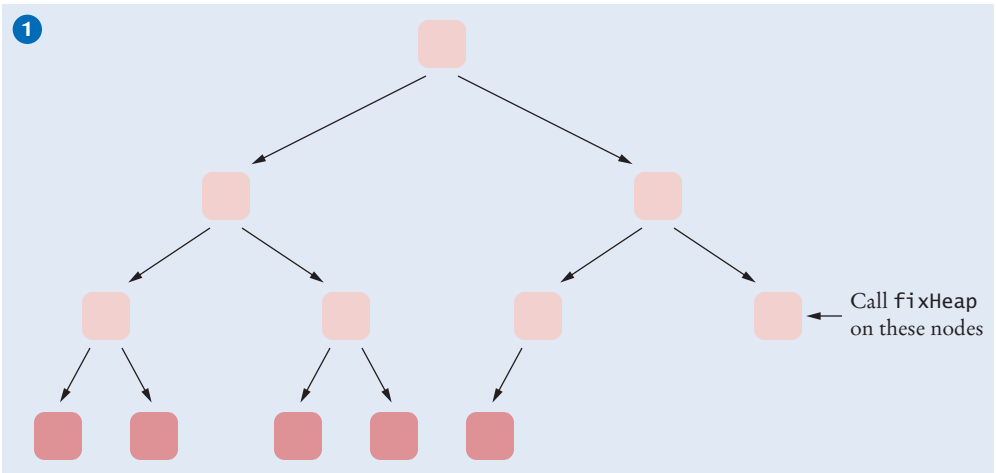


Figure 20 Turning a Tree into a Heap

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

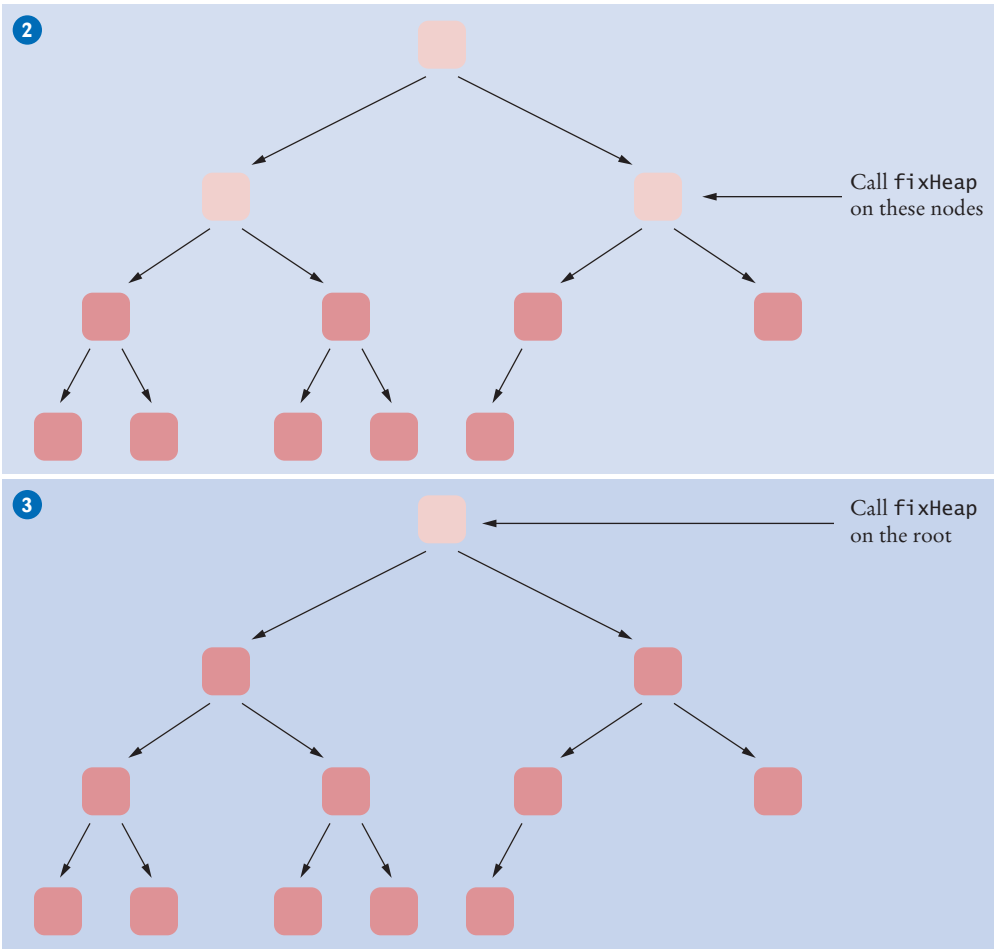


Figure 20 (continued) Turning a Tree into a Heap

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. Because we call the $O(\log(n))$ `fixHeap` method n times, this process requires $O(n \log(n))$ steps.

Rather than moving the root element into a separate array, we can *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).

```
while (n > 0)
{
   swap(0, n);
   n--;
   fixHeap(0, n);
}
```

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

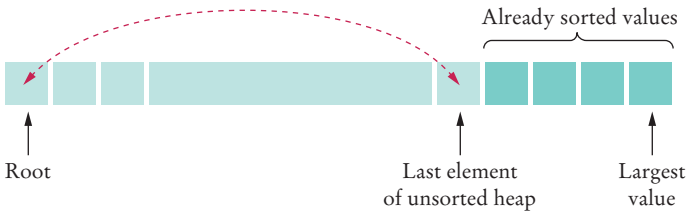


Figure 21 Using Heapsort to Sort an Array

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 21).

The following class implements the heapsort algorithm.

ch16/heapsort/HeapSorter.java

```
1  /**
2     This class applies the heapsort algorithm to sort an array.
3  */
4  public class HeapSorter
5  {
6     private int[] a;
7
8     /**
9        Constructs a heap sorter that sorts a given array.
10       @param anArray an array of integers
11    */
12    public HeapSorter(int[] anArray)
13    {
14       a = anArray;
15    }
16
17    /**
18       Sorts the array managed by this heap sorter.
19    */
20    public void sort()
21    {
22       int n = a.length - 1;
23       for (int i = (n - 1) / 2; i >= 0; i--)
24          fixHeap(i, n);
25       while (n > 0)
26       {
27          swap(0, n);
28          n--;
29          fixHeap(0, n);
30       }
31    }
32
```

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

```
33    /**
34       Ensures the heap property for a subtree, provided its
35       children already fulfill the heap property.
36       @param rootIndex the index of the subtree to be fixed
37       @param lastIndex the last valid index of the tree that
38       contains the subtree to be fixed
39    */
40    private void fixHeap(int rootIndex, int lastIndex)
41    {
42       // Remove root
43       int rootValue = a[rootIndex];
44
45       // Promote children while they are larger than the root
46
47       int index = rootIndex;
48       boolean more = true;
49       while (more)
50       {
51          int childIndex = getLeftChildIndex(index);
52          if (childIndex <= lastIndex)
53          {
54             // Use right child instead if it is larger
55             int rightChildIndex = getRightChildIndex(index);
56             if (rightChildIndex <= lastIndex
57                   && a[rightChildIndex] > a[childIndex])
58             {
59                childIndex = rightChildIndex;
60             }
61
62             if (a[childIndex] > rootValue)
63             {
64                // Promote child
65                a[index] = a[childIndex];
66                index = childIndex;
67             }
68             else
69             {
70                // Root value is larger than both children
71                more = false;
72             }
73          }
74          else
75          {
76             // No children
77             more = false;
78          }
79       }
80
81       // Store root value in vacant slot
82       a[index] = rootValue;
83    }
84
85    /**
86       Swaps two entries of the array.
87       @param i the first position to swap
88       @param j the second position to swap
89    */
90    private void swap(int i, int j)
91    {
```

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

```
92          int temp = a[i];
93          a[i] = a[j];
94          a[j] = temp;
95       }
96
97       /**
98          Returns the index of the left child.
99          @param index the index of a node in this heap
100         @return the index of the left child of the given node
101      */
102      private static int getLeftChildIndex(int index)
103      {
104         return 2 * index + 1;
105      }
106
107      /**
108         Returns the index of the right child.
109         @param index the index of a node in this heap
110         @return the index of the right child of the given node
111      */
112      private static int getRightChildIndex(int index)
113      {
114         return 2 * index + 2;
115      }
116   }
```

SELF CHECK

17. Which algorithm requires less storage, heapsort or merge sort?
18. Why are the computations of the left child index and the right child index in the `HeapSorter` different than in `MinHeap`?

Random Fact 16.1

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

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

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.

Summary of Learning Objectives

Describe the abstract set type and its implementations in the Java library.

- A set is an unordered collection of distinct elements. Elements can be added, located, and removed.
- Sets don't have duplicates. Adding a duplicate of an element that is already present is silently ignored.
- The `HashSet` and `TreeSet` classes both implement the `Set` interface.
- To visit all elements in a set, use an iterator.
- A set iterator visits elements in seemingly random order (`HashSet`) or sorted order (`TreeSet`).
- You cannot add an element to a set at an iterator position.

Describe the abstract map type and its implementations in the Java library.

- A map keeps associations between key and value objects.
- The `HashMap` and `TreeMap` classes both implement the `Map` interface.
- To find all keys and values in a map, iterate through the key set and find the values that correspond to the keys.

Explain the implementation of a hash table and its performance characteristics.

- A hash function computes an integer value from an object.
- A good hash function minimizes *collisions*—identical hash codes for different objects.

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

- A hash table can be implemented as an array of *buckets*—sequences of nodes that hold elements with the same hash code.
- If there are no or only a few collisions, then adding, locating, and removing hash table elements takes constant or $O(1)$ time.

Develop a `hashCode` method that is appropriate for a given class.

- When implementing the `hashCode` method, combine the hash codes for the instance variables.
- Your `hashCode` method must be compatible with the `equals` method.
- If a class provides neither `equals` nor `hashCode`, then objects are compared by identity.
- In a hash map, only the keys are hashed.

Explain the implementation of a binary search tree and its performance characteristics.

- A binary tree consists of nodes, each of which has at most two child nodes.
- 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.
- To insert a value into a binary search tree, keep comparing the value with the node data and follow the nodes to the left or right, until reaching a `null` node.
- When removing a node with only one child from a binary search tree, the child replaces the node to be removed.
- When removing a node with two children from a binary search tree, replace it with the smallest node of the right subtree.
- In a balanced tree, all paths from the root to the leaves have about the same length.
- Adding, finding, and removing an element in a tree set is proportional to the height of the tree.
- If a binary search tree is balanced, then adding, locating, or removing an element takes $O(\log(n))$ time.

Describe preorder, inorder, and postorder tree traversal.

- To visit all elements in a tree, visit the root and recursively visit the subtrees. We distinguish between preorder, inorder, and postorder traversal.
- Postorder traversal of an expression tree yields the instructions for evaluating the expression on a stack-based calculator.

Describe the behavior of the priority queue data type.

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

Describe the heap data structure and the efficiency of its operations.

- A heap is an almost completely filled tree in which the values of all nodes are at most as large as those of their descendants.
- Inserting or removing a heap element is an $O(\log(n))$ operation.
- The regular layout of a heap makes it possible to store heap nodes efficiently in an array.

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

Describe the heapsort algorithm and its run-time performance.

- The heapsort algorithm is based on inserting elements into a heap and removing them in sorted order.
- Heapsort is an $O(n \log(n))$ algorithm.

Classes, Objects, and Methods Introduced in this Chapter

java.util.Collection<E>
  contains
  remove
  size
java.util.HashMap<K, V>
java.util.HashSet<K, V>
java.util.Map<K, V>
  get
  keySet
  put
  remove
java.util.PriorityQueue<E>
  remove
java.util.Set<E>
java.util.TreeMap<K, V>
java.util.TreeSet<K, V>

Media Resources

www.wiley.com/college/horstmann

- ***Worked Example*** Word Frequency
- Lab Exercises
- Practice Quiz
- Code Completion Exercises

Review Exercises

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

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

★★ **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.)

★★ **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 666?

★★ **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.)

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

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

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

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

★ **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.

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

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

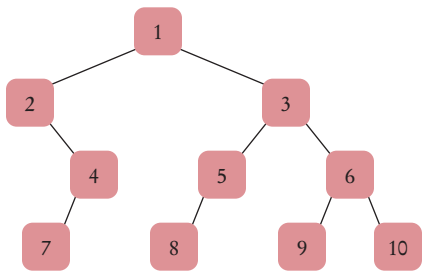
★ **R16.12** What is the difference between a balanced tree and an unbalanced tree? Give examples of each.

★ **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
Diana
Harry
```

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

★★ **R16.15** Consider the following tree. In which order are the nodes printed by the `BinarySearchTree.print` method? The numbers identify the nodes. The data stored in the nodes is not shown.



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

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

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

★★★ **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 $i/2$.

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

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

Show all steps.

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

Programming Exercises

★ **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.

★ **P16.2** Insert the 13 standard colors that the `Color` class declares (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.

★★ **P16.3** 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.

★ **P16.4** Insert all words from a large file (such as the novel "War and Peace", which is available on the Internet) into a hash set and a tree set. Time the results. Which data structure is faster?

★★★ **P16.5** Write a program that reads a Java source file and produces an index of all identifiers in the file. For each identifier, print all lines in which it occurs.
Hint: Call `in.useDelimiter("[^A-Za-z0-9_]+")`. Then each call to `next` returns a string consisting only of letters, numbers, and underscores.

★★ **P16.6** Try to find two words with the same hash code in a large file, such as the `/usr/share/dict/words` file on a Linux system. Keep a `Map<Integer, HashSet<String>>`. When you read in a word, compute its hash code h and put the word in the set whose key is h. Then iterate through all keys and print the sets whose size is > 1.

★★ **P16.7** Write a program that keeps a map in which both keys and values are strings—the names of students and their course grades. Prompt the user of the program to add or remove students, to modify grades, or to print all grades. The printout should be sorted by name and formatted like this:

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

★★★ **P16.8** Reimplement Exercise P16.7 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.

★★★ **P16.9** Add a `debug` method to the `HashSet` implementation in Section 16.4 that prints the nonempty buckets of the hash table. Run the test program at the end of Section 16.4. Call the `debug` method after all additions and removals and verify that Figure 6 accurately represents the state of the hash table.

★★ **P16.10** Supply compatible `hashCode` and `equals` methods to the `Student` class described in Exercise P16.8. Test the hash code by adding `Student` objects to a hash set.

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

★ **P16.11** 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.

★ **P16.12** A labeled point has *x*- and *y*-coordinates and a string label. Provide a class `LabeledPoint` with a constructor `LabeledPoint(int x, int y, String label)` and `hashCode` and `equals` methods. Two labeled points are considered the same when they have the same location and label.

★ **P16.13** Reimplement the `LabeledPoint` class of Exercise P16.12 by storing the location in a `java.awt.Point` object. Your `hashCode` and `equals` methods should call the `hashCode` and `equals` methods of the `Point` class.

★ **P16.14** Modify the `LabeledPoint` class of Exercise P16.13 so that it implements the `Comparable` interface. Sort points first by their *x*-coordinates. If two points have the same *x*-coordinate, sort them by their *y*-coordinates. If two points have the same *x*- and *y*-coordinates, sort them by their label. Write a tester program that checks all cases.

★★ **P16.15** 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 contains(int x)` to test whether `x` is present

★★ **P16.16** Reimplement the set class from Exercise P16.15 by using a `TreeSet<Integer>`. In addition to the methods specified in Exercise P16.15, 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.

★ **P16.17** Reimplement the set class from Exercise P16.15 by using a `TreeSet<Integer>`. In addition to the methods specified in Exercise P16.15, supply methods

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

that compute the union and intersection of two sets.

★ **P16.18** Write a method of the `BinarySearchTree` class

```
Comparable smallest()
```

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

★★★ **P16.19** 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.

★ **P16.20** Implement methods that use preorder and postorder traversal to print the elements in a binary search tree.

★★★ **P16.21** 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.

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

★★ **P16.22** Suppose an interface `Visitor` has a single method

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

Supply methods

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

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

★★ **P16.23** Apply Exercise P16.22 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.

★★ **P16.24** Modify the implementation of the `MinHeap` class so that the parent and child index positions and elements are computed directly, without calling helper methods.

★★★ **P16.25** Modify the implementation of the `MinHeap` class so that the `0` element of the array is not wasted.

★ **P16.26** Time the results of heapsort and merge sort. Which algorithm behaves better in practice?

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 16.7. (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. The words would be listed in sorted order.
4. When it is desirable to visit the set elements in sorted order.
5. A set stores elements. A map stores associations between keys and values.

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

6. The ordering does not matter, and you cannot have duplicates.
7. Yes, the hash set will work correctly. All elements will be inserted into a single bucket.
8. It locates the next bucket in the bucket array and points to its first element.
9. $31 \times 116 + 111 = 3707$.
10. 13.
11. 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.
12. For example, `Sarah`. Any string between `Romeo` and `Tom` will do.
13. For both trees, the inorder traversal is `3 + 4 * 5`.
14. 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`.
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.

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.