CHAPTER 14

SETS, MAPS, AND PRIORITY QUEUES

CHAPTER GOALS

To become familiar with the set, map, and priority queue data types

To understand the implementation of binary search trees and heaps

To learn about the efficiency of operations on tree structures

CHAPTER CONTENTS

If you want to write a program that collects objects (such as the stamps to the left), you have a number of choices. Of course, you can use an array or linked list, but computer scientists have invented other mechanisms that may be better suited for the task. In this chapter, you will learn how to use the set, map, and priority queue types that are provided in the C++ library. You will see how these data structures are implemented as tree-like structures, and how they trade off sequential ordering for fast element lookup.

14.1 Sets

Arrays, vectors and linked lists 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. You can then make a very useful tradeoff: Instead of keeping elements in order, you can find them quickly.

A set is an unordered collection of distinct elements.

In mathematics and computer science, an unordered collection of distinct items is called a *set*. As a typical example, consider a print server: a computer that has access to multiple printers. The server may keep a collection of objects representing available printers (see Figure 1). The order of the objects doesn't really matter.

The fundamental operations on a set are

- Adding an element.
- Removing an element.
- Finding an element.
- Traversing all elements .

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

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. 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 elements that are already in the set, as well as removing elements that are not in the set, are valid operations, but they do not change the set.

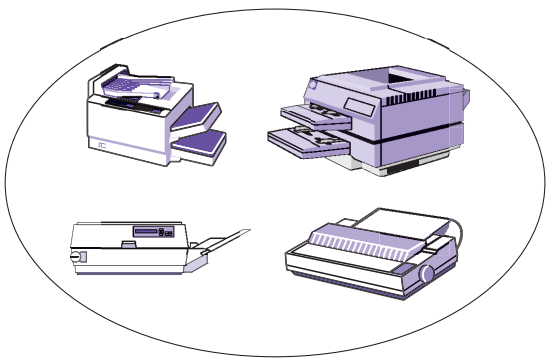


Figure 1 A Set of Printers

In C++, you use the `set` class to construct a set. As with vectors and lists, `set` requires a type parameter. For example, a set of strings is declared as follows:

```
set<string> names;
```

You use the `insert` and `erase` member functions to add and remove elements:

```
names.insert("Romeo");
names.insert("Juliet");
names.insert("Romeo"); // Has no effect: "Romeo" is already in the set
names.erase("Juliet");
names.erase("Juliet"); // Has no effect: "Juliet" is no longer in the set
```

To determine whether a value is in the set, use the `count` member function. It returns 1 if the value is in the set, 0 otherwise.

```
int c = names.count("Romeo"); // count returns 1
```

The standard C++ set class stores values in sorted order.

Finally, you can visit the elements of a set with an iterator. The iterator visits the elements in *sorted* order, not in the order in which you inserted them. For example, consider what happens when we continue our set example as follows:

```
names.insert("Tom");
names.insert("Diana");
names.insert("Harry");
set<string>::iterator pos;
for (pos = names.begin(); pos != names.end(); pos++)
{
   cout << *pos << " ";
}
```

The code prints the set elements in dictionary order:

```
Diana Harry Romeo Tom
```

A set cannot contain duplicates. A multiset (also called a bag) is an unordered collection that can contain multiple copies of an element. An example is a grocery bag that contains some grocery items more than once (see Figure 2).

A multiset (or bag) is similar to a set, but elements can occur multiple times.

In the C++ library, the `multiset` class implements this data type. You use a `multiset` in the same way as a set. When you insert an element multiple times, the element count reflects the number of insertions. Each call to `erase` decrements the element count until it reaches 0.

```
multiset<string> names;
names.insert("Romeo");
names.insert("Juliet");
names.insert("Romeo"); // Now names.count("Romeo") is 2
names.erase("Juliet"); // Now names.count("Juliet") is 0
names.erase("Juliet"); // Has no effect: "Juliet" is no longer in the bag
```

Figure 2
A Bag of Groceries

A good illustration of the use of sets is a program to check for misspelled words. Assume you have a file containing correctly spelled words (that is, a dictionary), and a second file you wish to check. The program reads a file containing a dictionary of correctly spelled words into a set, then reads words from the second file and tests each in the set, printing the word if it is not found.

```
void spell_check(istream& dictionary, istream& text)
{
   set<string> words;
   string word;

   // First put all words from the dictionary into the set
   while (dictionary >> word)
   {
      words.insert(word);
   }

   // Then read words from text
   while (text >> word)
   {
      if (words.count(word) == 0)
      {
         cout << "Misspelled word " << word << endl;
      }
   }
}
```

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. What is the output of the following code snippet?

```
set<int> s;
for (int i = 1; i <= 4; i++)
{
   s.insert(i * i);
   s.insert(-i);
}
for (set<int>::iterator p = s.begin(); p != s.end(); p++) { cout << *p << " "; }
```

4. What is the output of the following code snippet?

```
set<int> s;
for (int i = 0; i < 20; i = i + 2) { s.insert(i); }
for (int i = 0; i < 20; i = i + 3) { s.remove(i); }
for (set<int>::iterator p = s.begin(); p != s.end(); p++) { cout << *p << " "; }
```

5. What does the following code do, given two `set<string>` named `a` and `b`?

```
set<string> c;
for (set<string>::iterator p = a.begin(); p != a.end(); p++) { c.insert(*p); }
for (set<string>::iterator p = b.begin(); p != b.end(); p++)
{
   if (a.count(*p) == 0) { c.remove(*p); }
}
```

Practice It Now you can try these exercises at the end of the chapter: R14.1, R14.2, P14.3.

Special Topic 14.1

Defining an Ordering for Container Elements

The `set` and `multiset` classes need to compare elements. By default, these classes use the < operator for comparisons.

Suppose that you want to build a `set<Employee>`. The compiler will complain that it does not know how to compare two employees.

To solve this problem you can overload the < operator for `Employee` objects:

```
bool operator<(const Employee& a, const Employee& b)
{
   return a.get_name() < b.get_name();
}
```

This < operator compares employees by name. To learn more about overloading operators, see Horstmann and Budd, *Big C++, 2nd ed.,* Chapter 14 (John Wiley & Sons., Inc., 2009).

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

In the following sections we will introduce the simplest of many *tree* data structures that computer scientists have invented to overcome this problem.

14.2.1 Binary Trees and Binary Search Trees

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

A linked list is a one-dimensional data structure. You can imagine that all nodes are arranged in a line. In contrast, a *tree* is made of nodes that have children, which can again have children. You should visualize it as a tree, except that it is traditional to draw the tree upside down, like a family tree or hierarchy chart (see Figure 3). In a binary tree, every node has at most two children; hence the name *binary*.

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

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

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 3 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 child to the left, with data "Adam" before "Eve", and a single child to the right, with data "Harry" after "Eve". Check the remaining nodes in the same way.

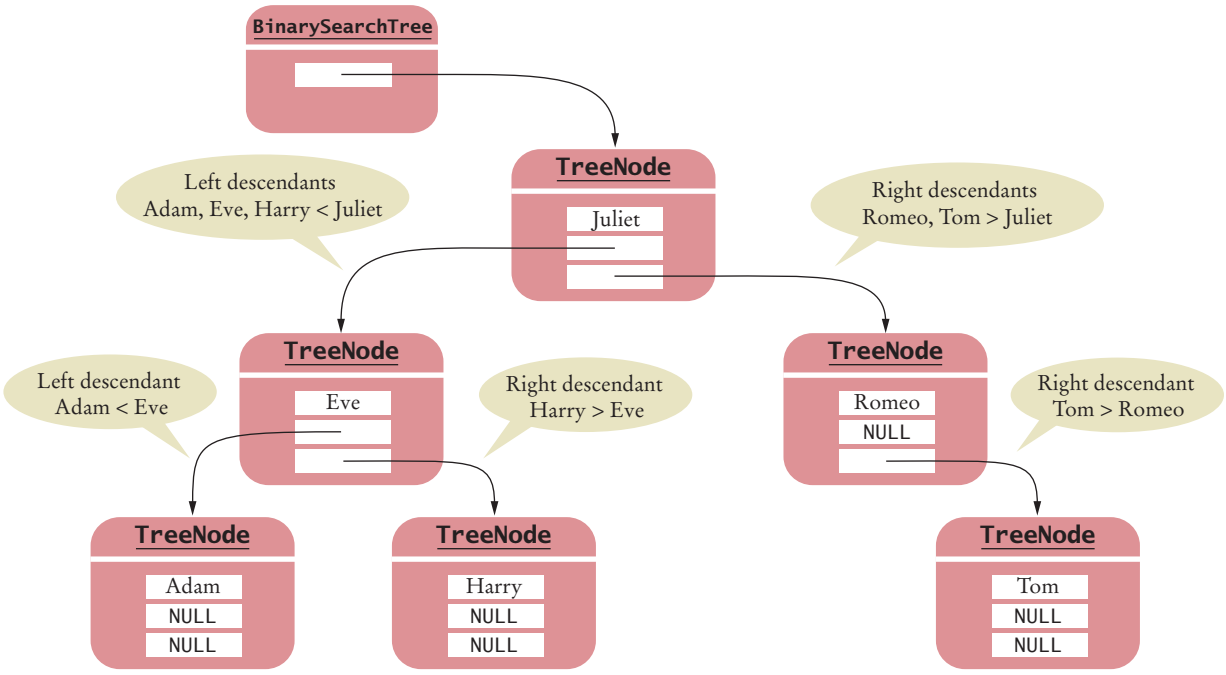


Figure 3 A Binary Search Tree

Figure 4 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.

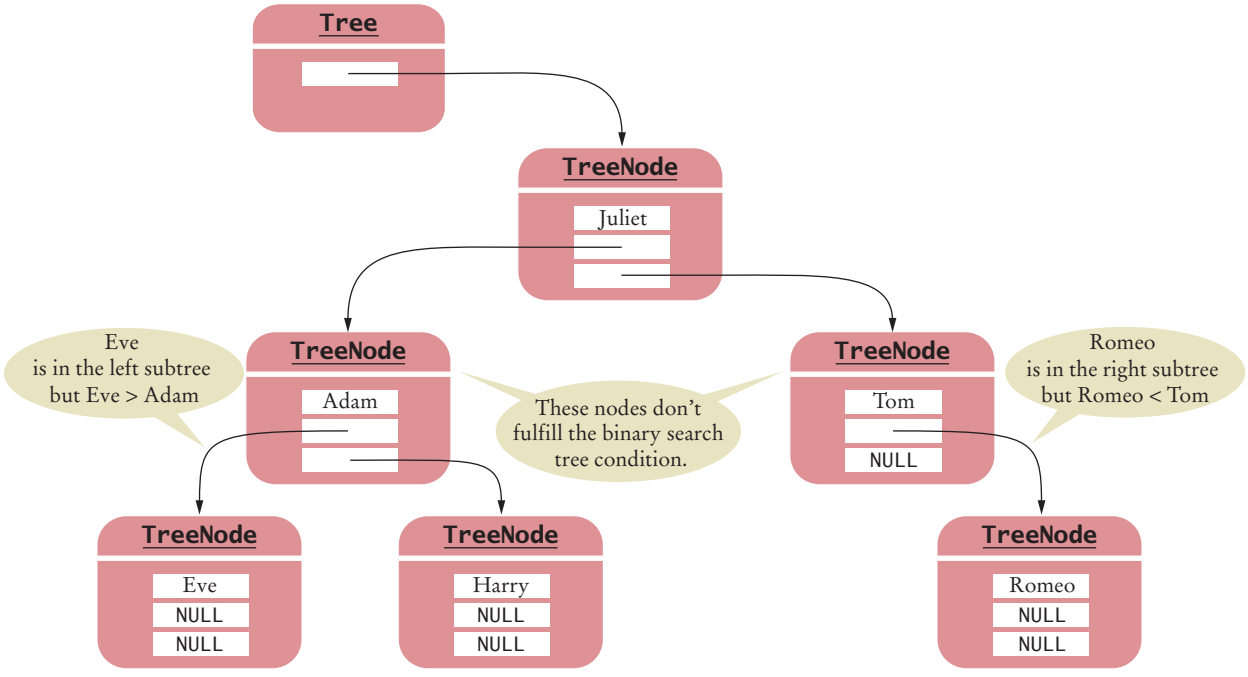


Figure 4 A Binary Tree That Is *Not* a Binary Search Tree

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 pointer to the *root node*, and a separate class for the nodes. Each node contains two pointers (to the left and right child nodes) and a data member. At the fringes of the tree, one or two of the child pointers are NULL.

```
class TreeNode
{
   ...
private:
   string data;
   TreeNode* left;
   TreeNode* right;
friend class BinarySearchTree;
};

class BinarySearchTree
{
   ...
private:
   TreeNode* root;
};
```

14.2.2 Inserting Elements into a Binary Search Tree

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

- When you encounter a node, 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 the left or right child is NULL, replace it with the new node.

For example, consider the tree in Figure 5. It is the result of the following statements:

```
BinarySearchTree;
tree.add("Juliet");
tree.add("Tom");
tree.add("Diana");
tree.add("Harry");
```

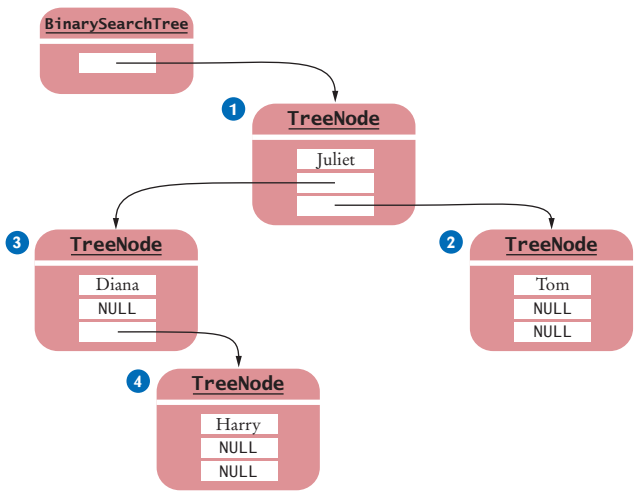


Figure 5
Binary Search Tree After Four Insertions

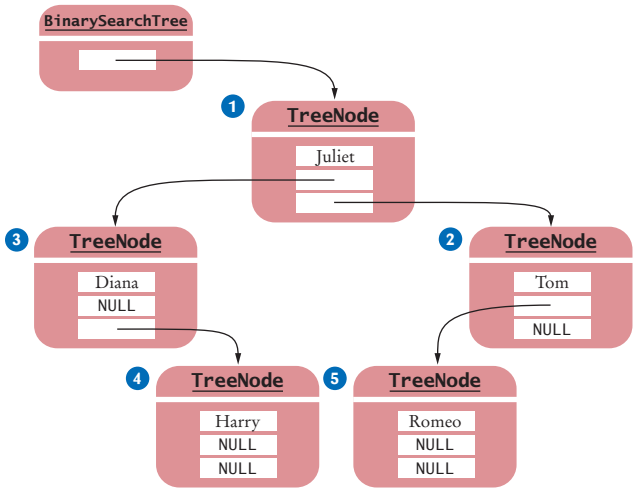


Figure 6 Binary Search Tree After Five Insertions

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

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

Start with the root, `Juliet`. `Romeo` comes after `Juliet`, so you move to the right subtree. You encounter the node `Tom`. `Romeo` comes before `Tom`, so you move to the left subtree. But there is no left subtree. Hence, you insert a new `Romeo` node as the left child of `Tom` (see Figure 6).

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 `insert` member function of the `BinarySearchTree` class:

```
void BinarySearchTree::insert(string data)
{
   TreeNode* new_node = new TreeNode;
   new_node->data = data;
   new_node->left = NULL;
   new_node->right = NULL;
   if (root == NULL)
   {
      root = new_node;
   }
   else
   {
      root->insert_node(new_node);
   }
}
```

To insert a value in a binary search tree, recursively insert it into the left or right subtree.

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 `insert_node` member function of the `TreeNode` class. That member function 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 it is larger than the object stored in the node, it is inserted in the right subtree:

```
void TreeNode::insert_node(TreeNode* new_node)
{
   if (new_node->data < data)
   {
      if (left == NULL)
      {
         left = new_node;
      }
      else
      {
         left->insert_node(new_node);
      }
   }
   else if (data < new_node->data)
   {
      if (right == NULL)
      {
         right = new_node;
      }
      else
      {
         right->insert_node(new_node);
      }
   }
}
```

Let us trace the calls to `insert_node` when inserting `Romeo` into the tree in Figure 5. The first call to `insert_node` is

```
root->insert_node(newNode)
```

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

```
root->right->insert_node(newNode)
```

The node `root->right` contains `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 `new_node`, and the insertion is complete (see Figure 6).

14.2.3 Removing Elements from a Binary Search Tree

Consider the task of removing 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.

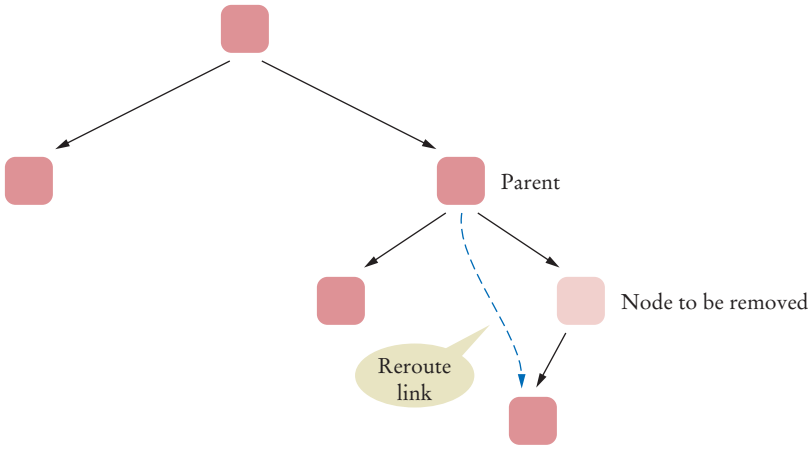


Figure 7 Removing a Node with One Child

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

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 7).

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

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

The case in which the node to be removed has two children is more challenging. Rather than removing the node, it is easier to replace its data value with the next larger value in the tree. That replacement preserves the binary search tree property. (Alternatively, you could use the largest element of the left subtree—see Exercise P14.11).

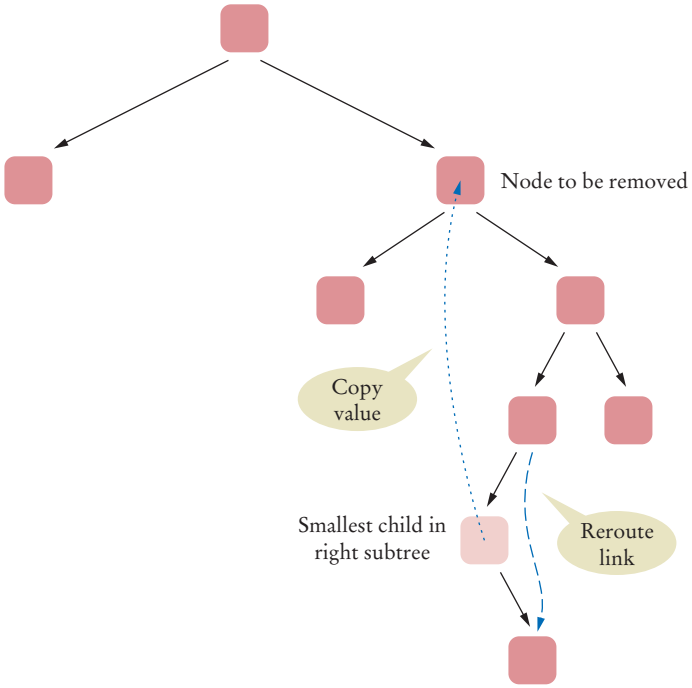


Figure 8
Removing a Node with Two Children

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

To locate the next larger value, go to the right subtree and find its smallest data value. Keep following the left child links. Once you reach a node that has no left child, you have found the node containing the smallest data value of the subtree. Now remove that node—it is easily removed because it has at most one child. Then store its data value in the original node that was slated for removal. Figure 8 shows the details.

You will find the complete source code for the `BinarySearchTree` class at the end of the next section. Now that you have seen how to implement this complex data structure, you may well wonder whether it is any good. Like nodes in a list, tree nodes are allocated one at a time. No existing elements need to be moved when a new element is inserted in the tree; that is an advantage. How fast insertion is, however, depends on the shape of the tree. If the tree is *balanced*—that is, if each node has approximately as many descendants on the left as on the right—then insertion takes $O(\log(n))$ time, where n is the number of nodes in the tree. This is a consequence of the fact that about half of the nodes are eliminated in each step. On the other hand, if the tree happens to be *unbalanced,* then insertion can be slow—perhaps as slow as insertion into a linked list (see Figure 9).

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

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

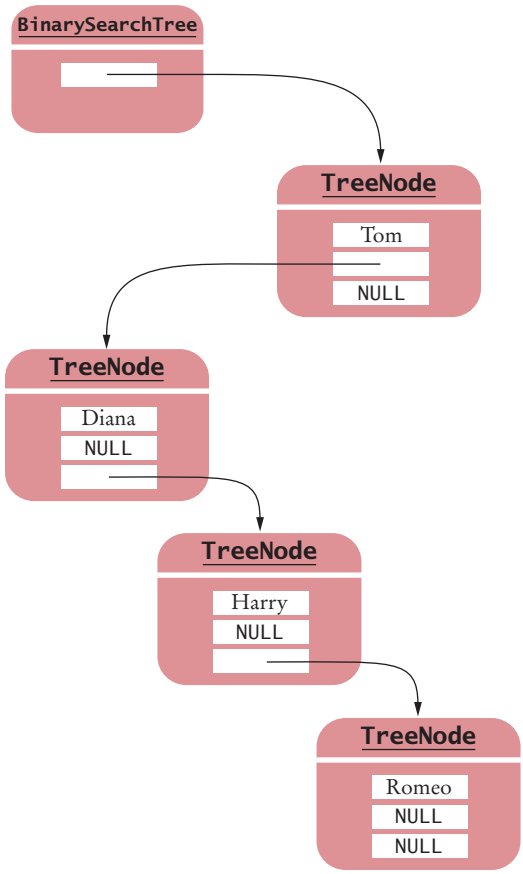


Figure 9 An Unbalanced Binary Search Tree

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

There are more sophisticated tree structures whose functions 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 C++ library uses *red-black trees*, a special form of balanced binary trees, to implement sets and maps.

Table 1 summarizes the performance of the fundamental operations on arrays or vectors, lists, and balanced binary trees.

Table 1 Execution Times for Container Operations

Operation	Array/ Vector	Linked List	Balanced Binary Tree
Add/remove element at end	$O(1)$	$O(1)$	N/A
Add/remove element in the middle	$O(n)$	$O(1)$	$O(\log(n))$
Get kth element	$O(1)$	$O(k)$	N/A
Find value	$O(n)$	$O(n)$	$O(\log(n))$

14.2.4 Tree Traversal

Once data has been inserted into a binary search tree, 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 node's data.
3. Print the right subtree.

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

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

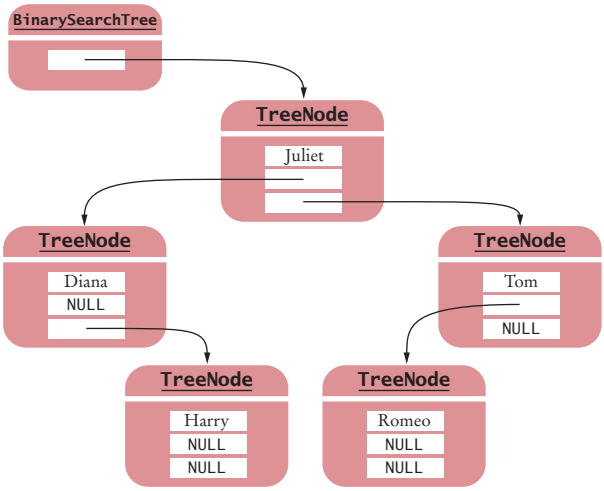


Figure 10 Traversing a Binary Search Tree

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.

Let us implement the `print` member function. You need a worker function `print_nodes` of the `TreeNode` class:

```
void TreeNode::print_nodes() const
{
   if (left != NULL)
   {
      left->print_nodes();
   }
   cout << data << endl;
   if (right != NULL)
   {
      right->print_nodes();
   }
}
```

To print the entire tree, start this recursive printing process at the root, with the following member function of the `BinarySearchTree` class:

```
void BinarySearchTree::print() const
{
   if (root != NULL)
   {
      root->print_nodes();
   }
}
```

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

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

In preorder traversal,

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

In postorder traversal,

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

These two traversals will not print the tree in sorted order. However, they are important in other applications of binary trees.

Tree traversals differ from an iterator in an important way. An iterator lets you visit a node at a time, and you can stop the iteration whenever you like. The traversals, on the other hand, visit all elements.

It turns out to be a bit complex to implement an iterator that visits the elements of a binary tree. Like a list iterator, a tree iterator contains a pointer to a node. The iteration starts at the leftmost element with no children (called a leaf). It then moves to the parent node, then to the right child, then to the next unvisited parent's leftmost child, and so on, until it reaches the rightmost element with no children. Exercise P14.12 and Exercise P14.13 discuss two methods for implementing such a tree iterator.

ch14/bintree.cpp

```
#include <iostream>
#include <string>

using namespace std;

class TreeNode
{
public:
   void insert_node(TreeNode* new_node);
   void print_nodes() const;
   bool find(string value) const;
private:
   string data;
   TreeNode* left;
   TreeNode* right;
friend class BinarySearchTree;
};

class BinarySearchTree
{
public:
   BinarySearchTree();
   void insert(string data);
   void erase(string data);
   int count(string data) const;
   void print() const;
private:
   TreeNode* root;
};
```

```
30
31  BinarySearchTree::BinarySearchTree()
32  {
33     root = NULL;
34  }
35
36  void BinarySearchTree::print() const
37  {
38     if (root != NULL)
39     {
40        root->print_nodes();
41     }
42  }
43
44  void BinarySearchTree::insert(string data)
45  {
46     TreeNode* new_node = new TreeNode;
47     new_node->data = data;
48     new_node->left = NULL;
49     new_node->right = NULL;
50     if (root == NULL)
51     {
52        root = new_node;
53     }
54     else
55     {
56        root->insert_node(new_node);
57     }
58  }
59
60  void TreeNode::insert_node(TreeNode* new_node)
61  {
62     if (new_node->data < data)
63     {
64        if (left == NULL)
65        {
66           left = new_node;
67        }
68        else
69        {
70           left->insert_node(new_node);
71        }
72     }
73     else if (data < new_node->data)
74     {
75        if (right == NULL)
76        {
77           right = new_node;
78        }
79        else
80        {
81           right->insert_node(new_node);
82        }
83     }
84  }
85
86  int BinarySearchTree::count(string data) const
87  {
88     if (root == NULL) { return 0; }
```

```
89       else if (root->find(data)) { return 1; }
90       else { return 0; }
91    }
92
93    void BinarySearchTree::erase(string data)
94    {
95       // Find node to be removed
96
97       TreeNode* to_be_removed = root;
98       TreeNode* parent = NULL;
99       bool found = false;
100      while (!found && to_be_removed != NULL)
101      {
102         if (to_be_removed->data < data)
103         {
104            parent = to_be_removed;
105            to_be_removed = to_be_removed->right;
106         }
107         else if (data < to_be_removed->data)
108         {
109            parent = to_be_removed;
110            to_be_removed = to_be_removed->left;
111         }
112         else found = true;
113      }
114
115      if (!found) return;
116
117      // to_be_removed contains data
118
119      // If one of the children is empty, use the other
120
121      if (to_be_removed->left == NULL || to_be_removed->right == NULL)
122      {
123         TreeNode* new_child;
124         if (to_be_removed->left == NULL)
125         {
126            new_child = to_be_removed->right;
127         }
128         else
129         {
130            new_child = to_be_removed->left;
131         }
132         if (parent == NULL) // Found in root
133         {
134            root = new_child;
135         }
136         else if (parent->left == to_be_removed)
137         {
138            parent->left = new_child;
139         }
140         else
141         {
142            parent->right = new_child;
143         }
144         return;
145      }
146
147      // Neither subtree is empty
148
```

```
149      // Find smallest element of the right subtree
150
151      TreeNode* smallest_parent = to_be_removed;
152      TreeNode* smallest = to_be_removed->right;
153      while (smallest->left != NULL)
154      {
155         smallest_parent = smallest;
156         smallest = smallest->left;
157      }
158
159      // smallest contains smallest child in right subtree
160
161      // Move contents, unlink child
162      to_be_removed->data = smallest->data;
163      if (smallest_parent == to_be_removed)
164      {
165         smallest_parent->right = smallest->right;
166      }
167      else
168      {
169         smallest_parent->left = smallest->right;
170      }
171   }
172
173   bool TreeNode::find(string value) const
174   {
175      if (value < data)
176      {
177         if (left == NULL)
178         {
179            return false;
180         }
181         else
182         {
183            return left->find(value);
184         }
185      }
186      else if (data < value)
187      {
188         if (right == NULL)
189         {
190            return false;
191         }
192         else
193         {
194            return right->find(value);
195         }
196      }
197      else
198      {
199         return true;
200      }
201   }
202
203   void TreeNode::print_nodes() const
204   {
205      if (left != NULL)
206      {
207         left->print_nodes();
208      }
```

```
209      cout << data << endl;
210      if (right != NULL)
211      {
212         right->print_nodes();
213      }
214   }
215
216   int main()
217   {
218      BinarySearchTree t;
219      t.insert("D");
220      t.insert("B");
221      t.insert("A");
222      t.insert("C");
223      t.insert("F");
224      t.insert("E");
225      t.insert("I");
226      t.insert("G");
227      t.insert("H");
228      t.insert("J");
229      t.erase("A"); // Removing element with no children
230      t.erase("B"); // Removing element with one child
231      t.erase("F"); // Removing element with two children
232      t.erase("D"); // Removing root
233      t.print();
234      cout << t.count("E") << endl;
235      cout << t.count("F") << endl;
236      return 0;
237   }
```

Program Run

```
C
E
G
H
I
J
1
0
```

6. What is the difference between a tree, a binary tree, and a balanced binary tree?
7. Give an example of a string that, when inserted into the tree of Figure 6, becomes a right child of `Romeo`.
8. Trace the call

   ```
   t.insert("C");
   ```

   in the `bintree.cpp` program of this section.
9. Give an example of seven strings that, when inserted into an empty binary search tree, yield a completely balanced tree. Give another example that yields a completely unbalanced tree.

10. Draw the tree that results from the following code:

```
BinarySearchTree t;
t.insert("H"); t.insert("E"); t.insert("L"); t.insert("L"); t.insert("O");
t.remove("W"); t.remove("O"); t.remove("R"); t.remove("L"); t.remove("D");
```

Practice It Now you can try these exercises at the end of the chapter: R14.8, R14.9, P14.8, P14.11.

14.3 Maps

A map keeps associations between key and value objects.

A map is a data type that keeps associations between *keys* and *values*. Every key in the map has a unique value, but a value may be associated with several keys. Figure 11 gives a typical example: a map that associates names with colors. This map might describe the favorite colors of various people.

With the `map` class in the standard library, you use the `[]` operator to associate keys and values. Here is an example:

```
map<string, double> scores;
scores["Tom"] = 90;
scores["Diana"] = 86;
scores["Harry"] = 100;
```

You can read a score back with the same notation:

```
cout << "Tom's score: " << scores["Tom"];
```

To find out whether a key is present in the map, use the `find` member function. It yields an iterator that points to the entry with the given key, or past the end of the container if the key is not present.

The iterator of a `map<K, V>` with key type `K` and value type `V` yields elements of type `pair<K, V>`. The `pair` class is a simple class defined in the `<utility>` header that stores a pair of values. It has two public (!) data members `first` and `second`.

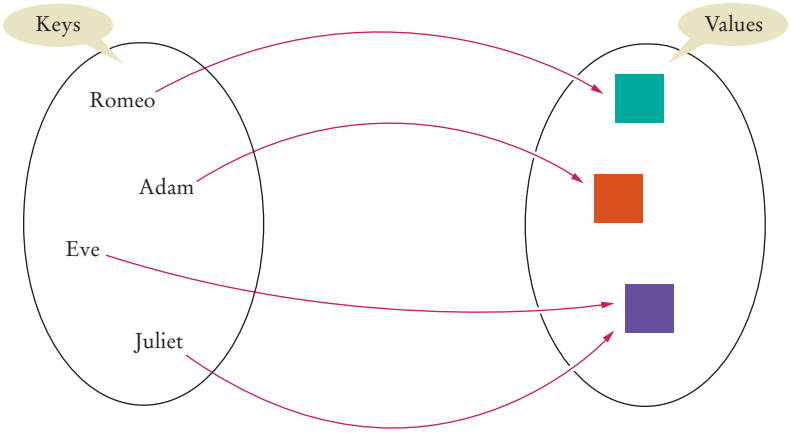


Figure 11 A Map

Therefore, you have to go through this process to see if a key is present:

```
map<string, double>::iterator pos = scores.find("Harry"); // Call find
if (pos == scores.end()) // Check if there was a match
{
   cout << "No match for Harry";
}
else
{
   cout << "Harry's score: " << (*pos).second;
      // pos points to a pair<string, double>
}
```

As with pointers, you can write `pos->second` instead of `(*pos).second`.

The following loop shows how you iterate over the contents of a map:

```
map<string, double>::iterator pos;
for (pos = scores.begin(); pos != scores.end(); pos++)
{
   cout << "The score of " << pos->first << " is " <<
      pos->second << endl;
}
```

A multimap can have multiple values associated with the same key.

A multimap can have multiple values associated with the same key. Instead of using the `[]` operator, you insert and erase pairs.

Here is an example:

```
multimap<string, string> friends;
friends.insert(make_pair("Tom", "Diana"));  // Diana is a friend of Tom
friends.insert(make_pair("Tom", "Harry"));  // Harry is also a friend of Tom
friends.erase(make_pair("Tom", "Diana"));  // Diana is no longer a friend of Tom
```

The `make_pair` function (also defined in the `<utility>` header) makes a `pair` object from its arguments.

To enumerate all values associated with a key, you obtain two iterators that define the range containing all pairs with a given key.

```
multimap<string, string>::iterator lower = friends.lower_bound("Tom");
multimap<string, string>::iterator upper = friends.upper_bound("Tom");
```

Then you visit all pairs in that range.

```
cout << "Tom's friends: ";
for (multimap<string, string>::iterator pos = lower; pos != upper; pos++)
{
   cout << pos->second << " ";
}
```

Maps and multimaps are implemented as binary trees whose nodes contain key/value pairs. The entries are ordered by increasing keys. You may need to define an `operator<` for the key type, as described in Special Topic 14.1.

A simple example to illustrate the use of maps and multimaps is a telephone database. The database associates names with telephone numbers. One member function inserts elements into the database. There are member functions to look up the number associated with a given name, and to carry out the inverse lookup of the names associated with a given number. Because two people can have the same number, we use a `multimap` for the inverse lookup. The member function `print_all` produces a listing of all entries. Because maps are stored in order based on their keys, this listing is naturally in alphabetical order according to name.

ch14/tele.cpp

```
#include <iostream>
#include <map>
#include <utility>
#include <string>
#include <vector>

using namespace std;

/**
   TelephoneDirectory maintains a map of name/number pairs
   and an inverse multimap of numbers and names.
*/
class TelephoneDirectory
{
public:
   /**
      Adds a new name/number pair to database.
      @param name the new name
      @param number the new number
   */
   void add_entry(string name, int number);

   /**
      Finds the number associated with a name.
      @param name the name being searched
      @return the associated number, or zero
      if not found in database
   */
   int find_entry(string name);

   /**
      Finds the names associated with a number.
      @param number the number being searched
      @return the associated names
   */
   vector<string> find_entries(int number);

   /**
      Prints all entries.
   */
   void print_all();
private:
   map<string, int> database;
   multimap<int, string> inverse_database;
};

void TelephoneDirectory::add_entry(string name, int number)
{
   database[name] = number;
   inverse_database.insert(make_pair(number, name));
}

int TelephoneDirectory::find_entry(string name)
{
   map<string, int>::iterator p = database.find(name);
   if (p == database.end())
   {
      return 0; // Not found
```

```
59       }
60       else
61       {
62          return p->second;
63       }
64    }
65
66    vector<string> TelephoneDirectory::find_entries(int number)
67    {
68       multimap<int, string>::iterator lower
69          = inverse_database.lower_bound(number);
70       multimap<int, string>::iterator upper
71          = inverse_database.upper_bound(number);
72       vector<string> result;
73
74       for (multimap<int, string>::iterator pos = lower;
75          pos != upper; pos++)
76       {
77          result.push_back(pos->second);
78       }
79       return result;
80    }
81
82    void TelephoneDirectory::print_all()
83    {
84       for (map<string, int>::iterator pos = database.begin();
85            pos != database.end(); pos++)
86       {
87          cout << pos->first << ": " << pos->second << endl;
88       }
89    }
90
91    int main()
92    {
93       TelephoneDirectory data;
94       data.add_entry("Fred", 7235591);
95       data.add_entry("Mary", 3841212);
96       data.add_entry("Sarah", 3841212);
97       cout << "Number for Fred: " << data.find_entry("Fred") << endl;
98       vector<string> names = data.find_entries(3841212);
99       cout << "Names for 3841212: ";
100      for (int i = 0; i < names.size(); i++)
101      {
102         cout << names[i] << " ";
103      }
104      cout << endl;
105      cout << "All names and numbers:" << endl;
106      data.print_all();
107      return 0;
108   }
```

Program Run

```
Number for Fred: 7235591
Names for 3841212: Mary Sarah
All names and numbers:
Fred: 7235591
Mary: 3841212
Sarah: 3841212
```

11. Define and initialize a map that maps the English words "one" through "five" to the numbers 1 through 5.
12. Write a loop that prints all values of the map that you defined in Self Check 11.
13. In which order are the values of Self Check 12 printed?
14. The index of a book specifies on which pages each term occurs. What C++ data structure can you use to model such an index?
15. Suppose you want to build a book index. Complete the following function that adds a word on a given page:

```
void Index::add_word(string word, int page)
{
   if (...) { word_pages[word] = set<int>(); }
   word_pages[word].insert(page);
}
```

Practice It Now you can try these exercises at the end of the chapter: R14.14, P14.1, P14.5.

Special Topic 14.2

Constant Iterators

If you carefully look at the source code of the `tele.cpp` program, you will notice that the member functions for finding and printing directory entries were not marked as `const`. If you properly implement them as constant member functions, the compiler will complain that the iterators are not constant. That is a legitimate problem because you can modify a container through an iterator.

Each iterator type has a companion type for a constant iterator, similar to a constant pointer. Here is a const-correct implementation of the `find_entry` function:

```
int TelephoneDirectory::find_entry(string name) const
{
   map<string, int>::const_iterator p = database.find(name);
   if (p == database.end())
   {
      return 0; // Not found
   }
   else
   {
      return p->second;
   }
}
```

14.4 Priority Queues

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

The final container we will examine is the priority queue. A *priority queue* is a container optimized for one special task; quickly locating the element with highest priority. Prioritization is a weaker condition than ordering. In a priority queue the order of the remaining elements is irrelevant, it is only the highest priority element that is important.

Consider this example, where a priority queue contains strings denoting tasks:

```
priority_queue<string> tasks;
tasks.push("2 - Shampoo carpets");
tasks.push("9 - Fix overflowing sink");
tasks.push("5 - Order cleaning supplies");
```

The strings are formatted so that they start with a priority number. When it comes time to do work, we will want to retrieve and remove the task with the top priority:

```
string task = tasks.top(); // Returns "9 - Fix overflowing sink"
tasks.pop();
```

The term priority *queue* is actually a misnomer, because the priority queue does not have the "first in/first out" behavior as does a true queue. In fact the interface for the priority queue is more similar to a stack than to a queue. The basic three operations are `push`, `pop`, and `top`. The `push` operation places a new element into the priority queue. `top` returns the element with highest priority; `pop` removes this element.

One obvious implementation for a priority queue is a sorted set. 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. Heaps store all elements in a single array, which is more efficient than storing each element in a tree node. We will describe heaps in the next section.

Here is a simple program that demonstrates a priority queue. Instead of storing strings, we use a `WorkOrder` class. As described in Special Topic 14.1, we supply an `operator<` function that compares work orders so that the priority queue can find the most important one.

ch14/pqueue.cpp

```
1  #include <iostream>
2  #include <queue>
3
4  using namespace std;
5
6  class WorkOrder
7  {
8  public:
9     WorkOrder(int priority, string description);
10    int get_priority() const;
11    string get_description() const;
12 private:
13    int priority;
14    string description;
15 };
16
17 WorkOrder::WorkOrder(int pr, string descr)
18 {
19    priority = pr;
20    description = descr;
21 }
22
23 int WorkOrder::get_priority() const
24 {
25    return priority;
26 }
27
```

```
28  string WorkOrder::get_description() const
29  {
30     return description;
31  }
32
33  bool operator<(WorkOrder a, WorkOrder b)
34  {
35     return a.get_priority() < b.get_priority();
36  }
37
38  int main()
39  {
40     priority_queue<WorkOrder> tasks;
41     tasks.push(WorkOrder(2, "Shampoo carpets"));
42     tasks.push(WorkOrder(3, "Empty trash"));
43     tasks.push(WorkOrder(2, "Water plants"));
44     tasks.push(WorkOrder(1, "Remove pencil sharpener shavings"));
45     tasks.push(WorkOrder(4, "Replace light bulb"));
46     tasks.push(WorkOrder(9, "Fix overflowing sink"));
47     tasks.push(WorkOrder(1, "Clean coffee maker"));
48     tasks.push(WorkOrder(5, "Order cleaning supplies"));
49
50     while (tasks.size() > 0)
51     {
52        WorkOrder task = tasks.top();
53        tasks.pop();
54        cout << task.get_priority() << " - "
55           << task.get_description() << endl;
56     }
57     return 0;
58  }
```

Program Run

```
9 - Fix overflowing sink
5 - Order cleaning supplies
4 - Replace light bulb
3 - Empty trash
2 - Water plants
2 - Shampoo carpets
1 - Remove pencil sharpener shavings
1 - Clean coffee maker
```

16. 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 C++ container type is appropriate for this application?
17. How could we have implemented the `pqueue.cpp` program of this section with a map?
18. What is the advantage of using a priority queue instead of a map of priority values to tasks?

Practice It Now you can try these exercises at the end of the chapter: R14.15, P14.14.

Special Topic 14.3

Discrete Event Simulations

A classic application of priority queues is in a type of simulation called a *discrete event simulation*. An event has a time at which it is scheduled to occur, and an action.

```
class Event
{
public:
   double get_time() const;
   virtual void act();
   ...
};
```

You form derived classes of the `Event` class for each event type. For example, an `Arrival` event can indicate the arrival of a customer, and a `Departure` event can indicate that the customer is departing. Each derived class overrides the `act` function. The `act` function is called at the time for which the event is scheduled.

Consider the `act` function of a `Departure` event. It will remove the customer from its current position (which might be a table or a cash register, depending on the simulation). The position is now available for another customer. The `act` function will locate such a customer (perhaps in a queue of waiting customers). Finally, the `act` function generates a `Departure` event for that customer at some, usually random, time in the future. In this way, new events are generated as the simulation evolves.

Now consider an `Arrival` event. Its `act` function deals with the fact that a new customer is arriving. If there is room to service the customer, then the `act` function should place the customer at a free position (such as a table or cash register) and schedule a `Departure` event for that customer at some time in the future. Moreover, in order to simulate a steady stream of customers, the `act` function should also schedule an `Arrival` event for another customer.

The heart of the simulation is the event loop. This loop pulls the next event from the priority queue of waiting events. Two events are compared based on their time. The comparison is inverted, so that the element with highest priority is the one with the lowest scheduled time. Events can be inserted in any order, but are removed in sequence based on their time. As each event is removed, the "system clock" advances to the event's time, and the virtual `act` function of the event is executed:

```
while (event_queue.size() > 0)
{
   Event* next_event = event_queue.top();
   current_time = next_event->get_time();
   next_event->act(); // Typically adds new events
   delete next_event;
}
```

We face a technical issue when defining the event queue. The event queue holds `Event*` pointers that point to instances of derived classes. Since pointers already have a `<` operator defined, we cannot define an `operator<` that compares `Event*` pointers by their timestamp. Instead, we define a function for this purpose:

```
bool event_less(const Event* e1, const Event* e2)
{
   return e1->get_time() > e2->get_time();
      // The earliest event should have the largest priority
}
```

We then tell the priority queue to use this comparison function:

```
priority_queue<Event*, vector<Event*>,
   bool (*)(const Event*, const Event*)> event_queue(event_less);
```

Exercise P14.16 asks you to simulate customers in a bank. Such simulations are important in practice because they give valuable information to business managers. For example, suppose

you expect 60 customers per hour, each of whom needs to see a teller for an average of 5 minutes. Hiring 5 tellers should be enough to service all customers, but if you run the simulation, you may find that the average customer has to wait in line about 10 minutes. By running simulations, you can determine tradeoffs between unhappy customers and idle tellers.

14.5 Heaps

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

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

1. A heap is *almost complete:* all nodes are filled in, except the last level may have some nodes missing toward the right (see Figure 12).
2. The tree fulfills the *heap property:* all nodes store values that are at least as large as the values stored in their descendants (see Figure 13).

It is easy to see that the heap property ensures that the tree's largest element is stored in the root.

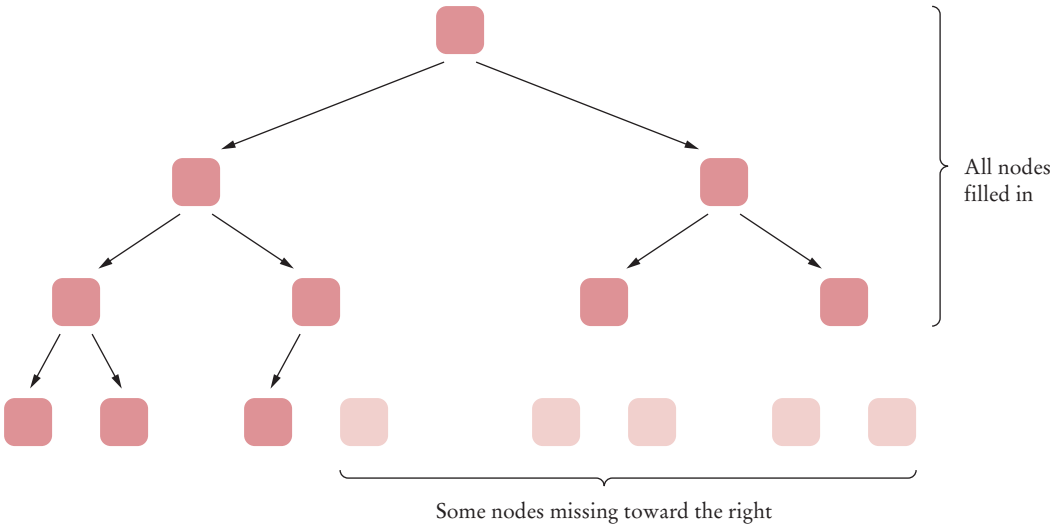


Figure 12
An Almost Complete Tree

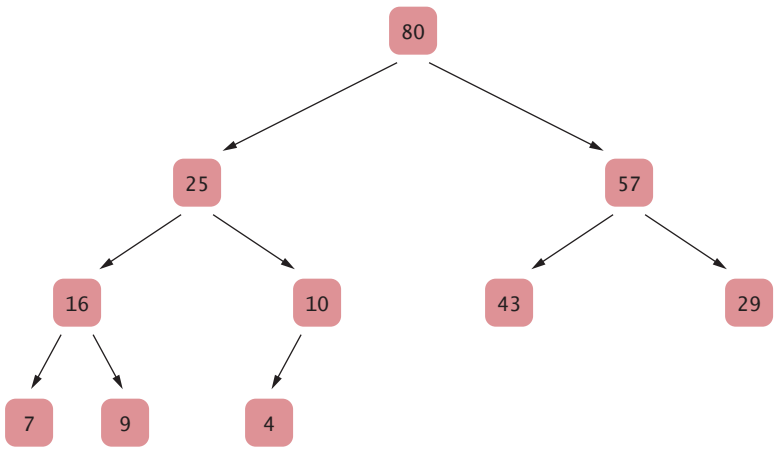


Figure 13
A Heap

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

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

Suppose we have a heap and want to insert a new element. After the insertion, the heap property should again be fulfilled. The following algorithm carries out the insertion (see Figure 14).

1. First, add a vacant slot to the end of the tree.
2. Next, demote the parent of the empty slot if it is smaller 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 smaller than the element to be inserted.
3. At this point, either the vacant slot is at the root, or the parent of the vacant slot is larger than the element to be inserted. Insert the element into the vacant slot.

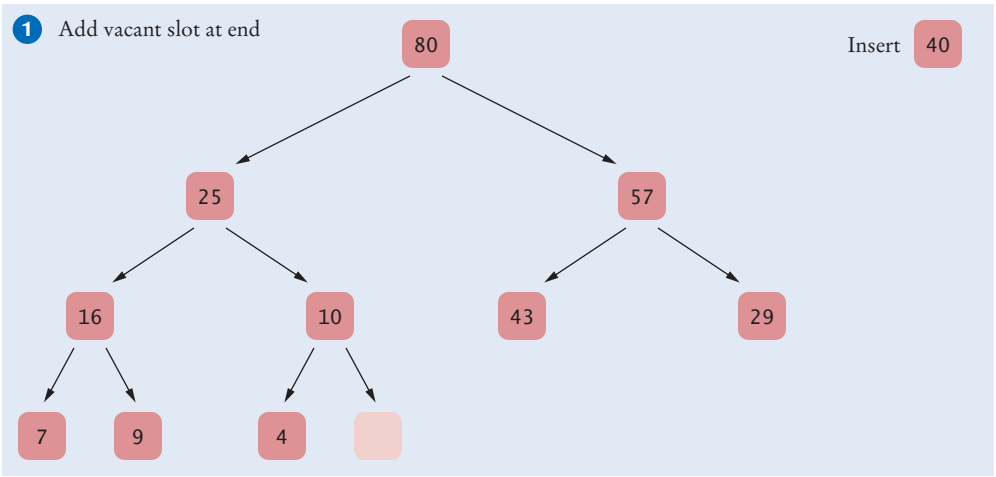


Figure 14 Inserting an Element into a Heap

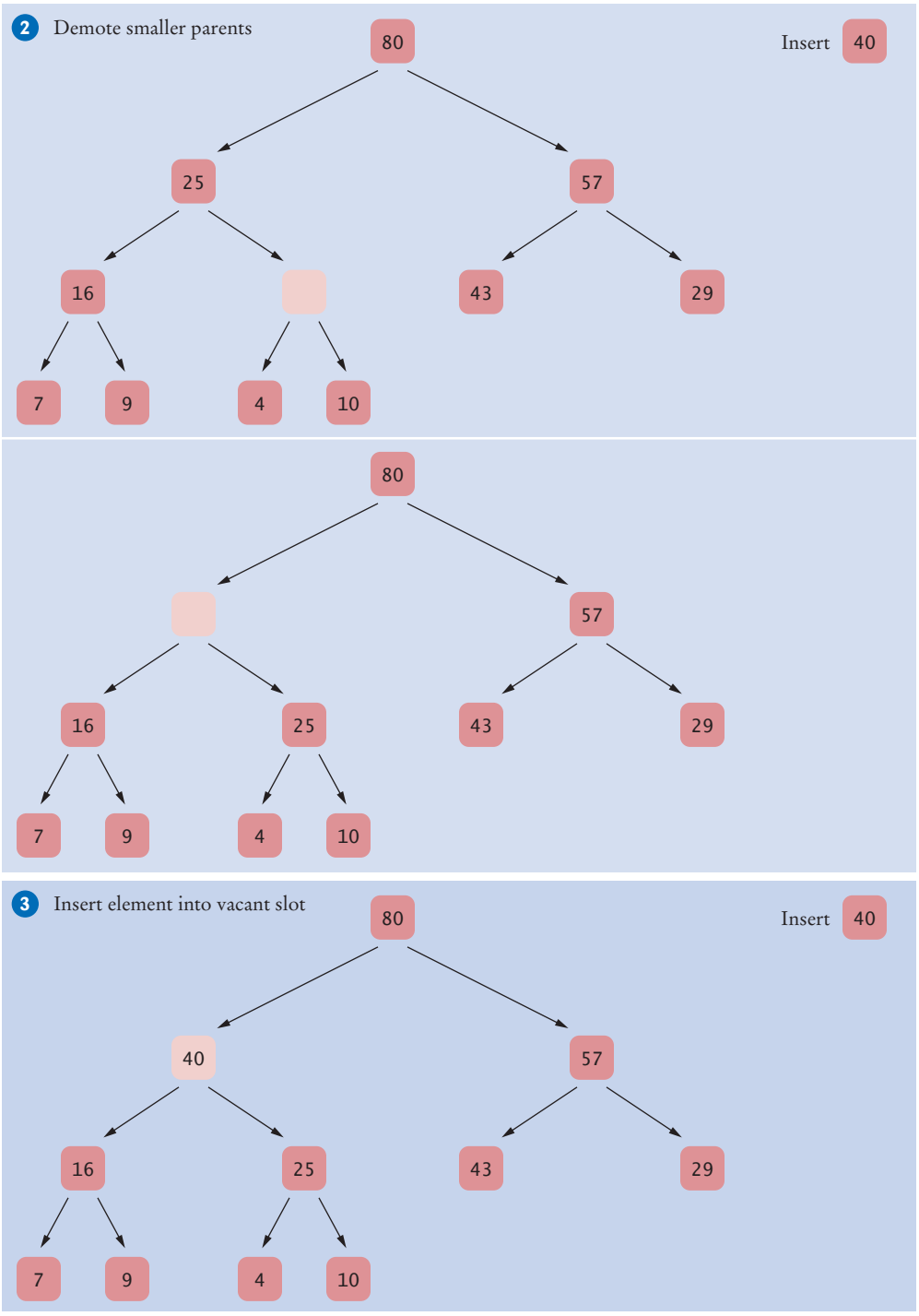


Figure 14 (continued) Inserting an Element into a Heap

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 maximum of all of the values in the heap.

Figure 15 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 larger.
3. Promote the larger child of the root node. (See Figure 15 continued.) Now the root node again fulfills the heap property. Repeat this process with the demoted child. That is, promote the larger of its children. Continue until the demoted child has no larger children. The heap property is now fulfilled again. This process is called "fixing the heap".

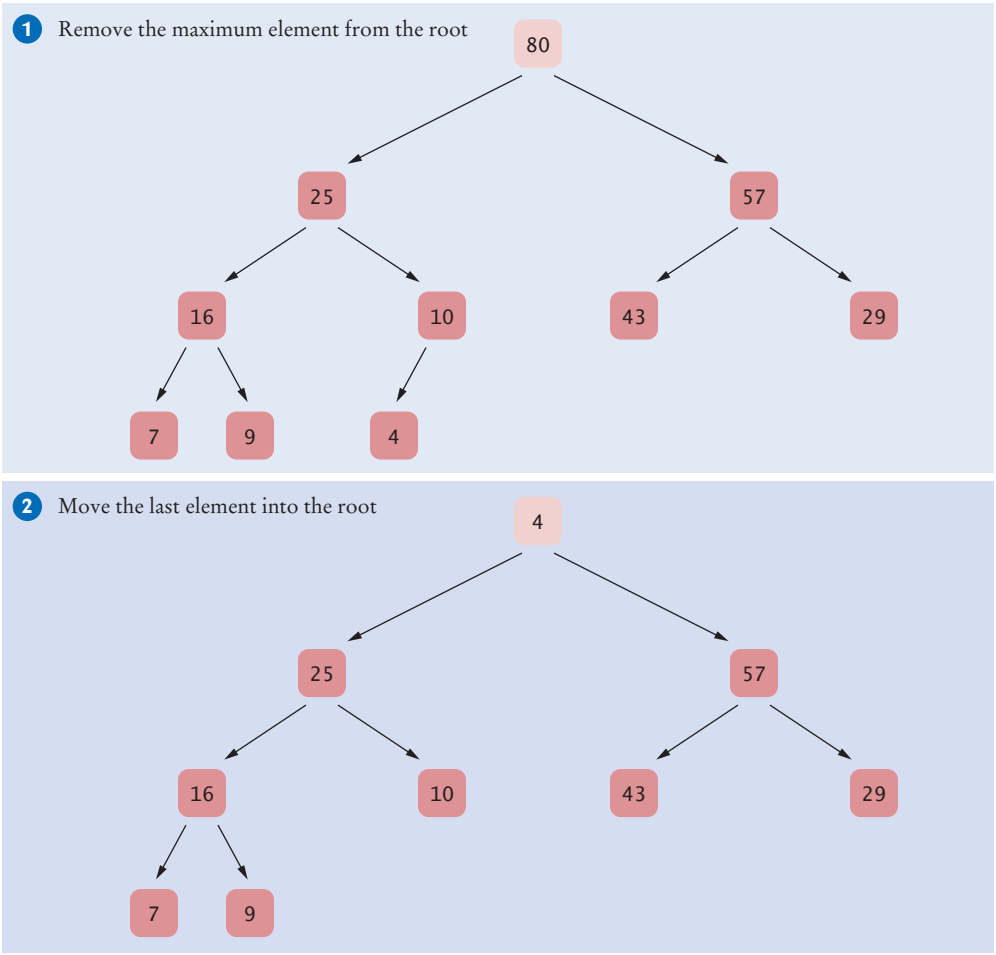


Figure 15 Removing the Maximum Element from a Heap

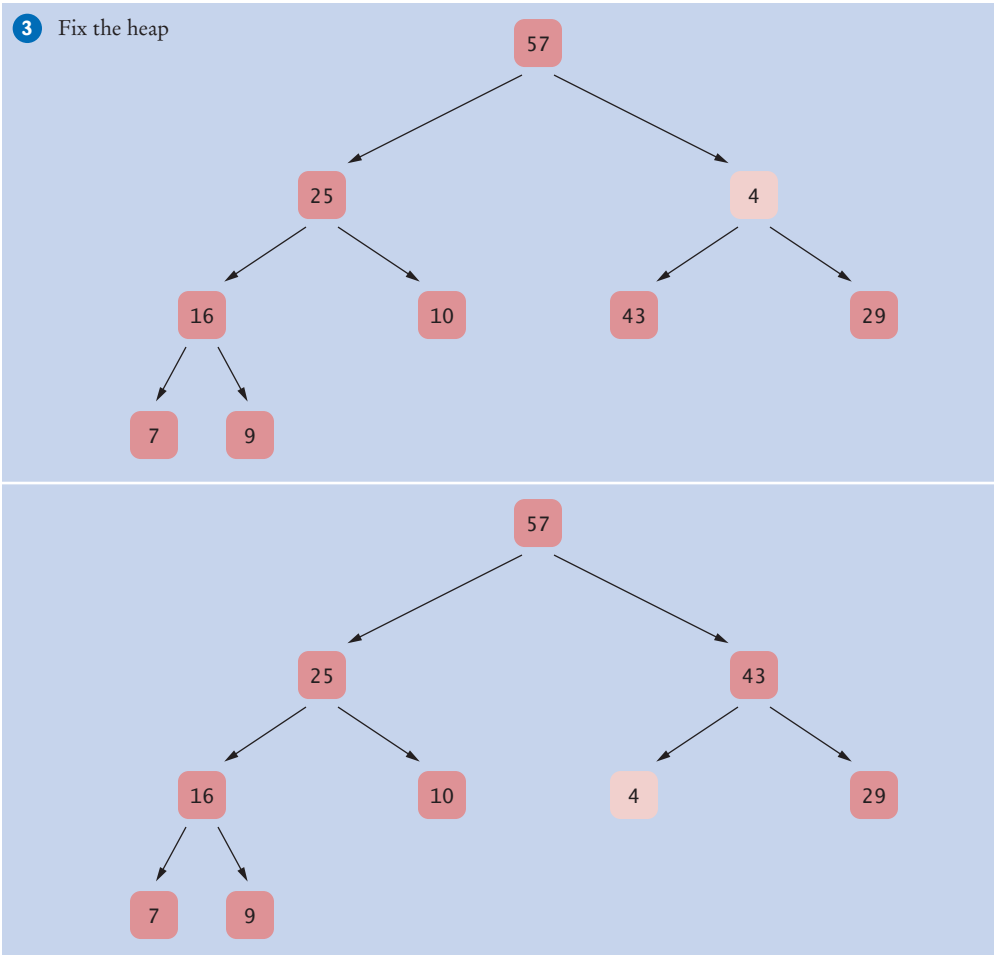


Figure 15 (continued) Removing the Maximum Value from a Heap

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.

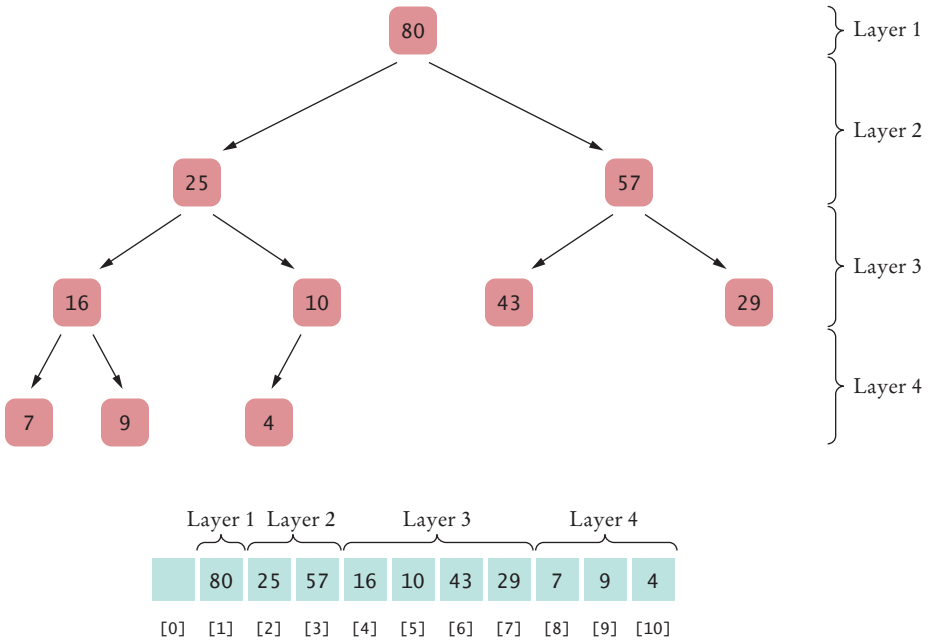


Figure 16 Storing a Heap in an Array

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

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

Heaps have another major advantage. Because of the regular layout of the heap nodes, it is easy to store the node values in an array. First store the first layer, then the second, and so on (see Figure 16). 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 16, 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 of integers. Using templates, it is easy to extend the class to a heap of any ordered type. (See Horstmann and Budd, *Big C++, 2nd ed.*, Chapter 16, for more information about templates.)

ch14/heap.cpp

```
1  #include <iostream>
2  #include <vector>
3
4  using namespace std;
5
6  /**
7     This class implements a heap.
8  */
9  class Heap
10 {
```

```
11  public:
12     /**
13        Constructs an empty heap.
14     */
15     Heap();
16
17     /**
18        Adds a new element to this heap.
19        @param new_element the element to add
20     */
21     void push(int new_element);
22
23     /**
24        Gets the maximum element stored in this heap.
25        @return the maximum element
26     */
27     int top() const;
28
29     /**
30        Removes the maximum element from this heap.
31     */
32     void pop();
33
34     /**
35        Returns the number of elements in this heap.
36     */
37     int size() const;
38  private:
39     /**
40        Turns the tree back into a heap, provided only the root
41        node violates the heap condition.
42     */
43     void fix_heap();
44
45     /**
46        Returns the index of the left child.
47        @param index the index of a node in this heap
48        @return the index of the left child of the given node
49     */
50     int get_left_child_index(int index);
51
52     /**
53        Returns the index of the right child.
54        @param index the index of a node in this heap
55        @return the index of the right child of the given node
56     */
57     int get_right_child_index(int index);
58
59     /**
60        Returns the index of the parent.
61        @param index the index of a node in this heap
62        @return the index of the parent of the given node
63     */
64     int get_parent_index(int index);
65
66     /**
67        Returns the value of the left child.
68        @param index the index of a node in this heap
69        @return the value of the left child of the given node
70     */
```

```
71       int get_left_child(int index);
72
73       /**
74          Returns the value of the right child.
75          @param index the index of a node in this heap
76          @return the value of the right child of the given node
77       */
78       int get_right_child(int index);
79
80       /**
81          Returns the value of the parent.
82          @param index the index of a node in this heap
83          @return the value of the parent of the given node
84       */
85       int get_parent(int index);
86
87       vector<int> elements;
88    };
89
90    Heap::Heap()
91    {
92       elements.push_back(0);
93    }
94
95    void Heap::push(int new_element)
96    {
97       // Add a new element
98       elements.push_back(0);
99       int index = elements.size() - 1;
100
101      // Demote parents that are smaller than the new element
102      while (index > 1
103         && get_parent(index) < new_element)
104      {
105         elements[index] = get_parent(index);
106         index = get_parent_index(index);
107      }
108
109      // Store the new element into the vacant slot
110      elements[index] = new_element;
111   }
112
113   int Heap::top() const
114   {
115      return elements[1];
116   }
117
118   void Heap::pop()
119   {
120      // Remove last element
121      int last_index = elements.size() - 1;
122      int last = elements[last_index];
123      elements.pop_back();
124
125      if (last_index > 1)
126      {
127         elements[1] = last;
128         fix_heap();
129      }
130   }
```

```
131
132 int Heap::size() const
133 {
134    return elements.size() - 1;
135 }
136
137 void Heap::fix_heap()
138 {
139    int root = elements[1];
140
141    int last_index = elements.size() - 1;
142    // Promote children of removed root while they are larger than last
143
144    int index = 1;
145    bool more = true;
146    while (more)
147    {
148       int child_index = get_left_child_index(index);
149       if (child_index <= last_index)
150       {
151          // Get larger child
152
153          // Get left child first
154          int child = get_left_child(index);
155
156          // Use right child instead if it is larger
157          if (get_right_child_index(index) <= last_index
158             && get_right_child(index) > child)
159          {
160             child_index = get_right_child_index(index);
161             child = get_right_child(index);
162          }
163
164          // Check if smaller child is larger than root
165          if (child > root)
166          {
167             // Promote child
168             elements[index] = child;
169             index = child_index;
170          }
171          else
172          {
173             // Root is larger than both children
174             more = false;
175          }
176       }
177       else
178       {
179          // No children
180          more = false;
181       }
182    }
183
184    // Store root element in vacant slot
185    elements[index] = root;
186 }
187
188 int Heap::get_left_child_index(int index)
189 {
190    return 2 * index;
```

```
191 }
192
193 int Heap::get_right_child_index(int index)
194 {
195    return 2 * index + 1;
196 }
197
198 int Heap::get_parent_index(int index)
199 {
200    return index / 2;
201 }
202
203 int Heap::get_left_child(int index)
204 {
205    return elements[2 * index];
206 }
207
208 int Heap::get_right_child(int index)
209 {
210    return elements[2 * index + 1];
211 }
212
213 int Heap::get_parent(int index)
214 {
215    return elements[index / 2];
216 }
217
218 int main()
219 {
220    Heap tasks;
221    tasks.push(2);
222    tasks.push(3);
223    tasks.push(2);
224    tasks.push(1);
225    tasks.push(4);
226    tasks.push(9);
227    tasks.push(1);
228    tasks.push(5);
229
230    while (tasks.size() > 0)
231    {
232       int task = tasks.top();
233       tasks.pop();
234       cout << task << endl;
235    }
236    return 0;
237 }
```

Program Run

```
9
5
4
3
2
2
1
1
```

19. 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`?

20. Consider the following sorting algorithm. Let `a` be an array with *n* elements. In the first phase, insert `a[0]`, `a[i]`, and so on, into a heap that is stored in the same array. (After *k* insertions, the heap occupies the first *k* elements of the array, and the elements that have yet to be inserted occupy the remaining *n* – *k* elements.) In the second phase, keep removing the largest element from the heap and store it in the vacant position at the end of the heap, that is, at `a[n - 1]`, `a[n - 2]`, and so on. What is the efficiency of this algorithm?

21. What advantage does the algorithm described in Self Check 20 have over the merge sort algorithm?

22. Consider the heap

```
                 10
         9                8
    7         6       5         4
 3    2    1
```

What is the heap when the value 11 is inserted?

23. Consider the heap

```
                 10
         9                8
    7         6       5         4
 3    2    1
```

What is the heap when the maximum is removed?

24. What binary trees correspond to the arrays 1 2 3 4 5 6 7 8 9 10 and 10 9 8 7 6 5 4 3 2 1? Which of them is a max-heap?

Practice It Now you can try these exercises at the end of the chapter: P14.17, P14.18.

CHAPTER SUMMARY

Describe the set data type and its implementation in the C++ library.

- A set is an unordered collection of distinct elements.
- Sets don't have duplicates. Adding a duplicate of an element that is already present is ignored.
- The standard C++ `set` class stores values in sorted order.
- A multiset (or bag) is similar to a set, but elements can occur multiple times.

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 in a binary search tree, recursively insert it into the left or right subtree.

- 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.
- If a binary search tree is balanced, then inserting an element takes $O(\log(n))$ time.
- Tree traversal schemes include preorder traversal, inorder traversal, and postorder traversal.

Describe the map data type and its implementation in the C++ library.

- A map keeps associations between key and value objects.
- A multimap can have multiple values associated with the same key.

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 complete tree in which the values of all nodes are at least 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.

REVIEW EXERCISES

R14.1 A school web site keeps a collection of web sites that are blocked at student computers. Should the program that checks for blocked sites use a vector, linked list, list, or set for storing the site addresses?

R14.2 A library wants to track which books are checked out to which patrons. Should they use a map or a multimap from books to patrons?

R14.3 A library wants to track which patrons have checked out which books. Should they use a map or a multimap from patrons to books?

R14.4 In an emergency, a case record is made for each incoming patient that describes the severity of the case. When doctors become available, they handle the most severe cases first. Should the case records be stored in a set, a map, or a priority queue?

R14.5 You keep a set of `Point` objects for a scientific experiment. (A `Point` has x and y coordinates.) Define a suitable `operator<` so that you can form a `set<Point>`.

R14.6 A `set<T>` can be implemented as a binary tree whose nodes store data of type `T`. How can you implement a `multiset<T>`?

R14.7 What is the difference between a binary tree and a binary search tree? Give examples of each.

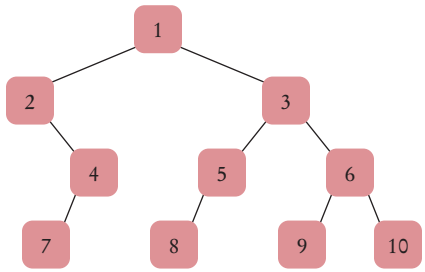
R14.8 What is the difference between a balanced tree and an unbalanced tree? Give examples of each.

R14.9 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
```

R14.10 Insert the elements of Exercise R14.9 in opposite order. Then determine how the `BinarySearchTree.print` function prints out both the tree from Exercise R14.9 and this tree. Explain how the printouts are related.

R14.11 Consider the following tree. In which order are the nodes printed by the `BinarySearchTree.print` function?



R14.12 How does a set achieve fast execution for insertions and removals?

R14.13 What properties of a binary tree make it a search tree? What properties make it a balanced tree?

R14.14 How is a map similar to a vector? How is it different?

R14.15 Why is a priority queue not, properly speaking, a queue?

R14.16 Prove that a heap of height h contains at least 2^{h-1} elements but less than 2^h elements.

R14.17 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 $i/2$.

PROGRAMMING EXERCISES

P14.1 Reimplement the `Polynomial` class of Exercise P13.14 by using a `map<int, double>` to store the coefficients.

P14.2 Write functions

```
set<int> set_union(set<int> a, set<int> b)
set<int> intersection(set<int> a, set<int> b)
```

that compute the set union and intersection of the sets `a` and `b`. (Don't name the first function `union`—that is a reserved word in C++.)

P14.3 Implement the *sieve of Eratosthenes:* a function for computing prime numbers, known to the ancient Greeks. Choose an integer n. This function will compute all prime numbers up to n. First insert all numbers from 1 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 $\sqrt{n}$. The remaining numbers are all primes.

P14.4 Write a program that counts how often each word occurs in a text file. Use a `multiset<string>`.

P14.5 Repeat Exercise P14.4, but use a `map<string, int>`.

P14.6 Write a member function of the `BinarySearchTree` class

```
string smallest()
```

that returns the smallest element of a tree.

P14.7 Change the `BinarySearchTree.print` member function to print the tree as a tree shape. It is easier to print the tree sideways. Extra credit if you instead print the tree with the root node centered on the top.

P14.8 Implement member functions that use preorder and postorder traversal to print the elements in a binary search tree.

P14.9 Implement a traversal function

```
void inorder(Action& a);
```

for inorder traversal of a binary search tree that carries out an action other than just printing the node data. The action should be supplied as a derived class of the class

```
class Action
{
public:
   void act(string str);
};
```

P14.10 Use the `inorder` function of Exercise P14.9, and a suitable class derived from `Action`, to compute the sum of all lengths of the strings stored in a tree.

P14.11 In the `BinarySearchTree` class, modify the `erase` member function so that a node with two children is replaced by the largest child of the left subtree.

P14.12 Add a pointer to the parent node to the `TreeNode` class. Modify the `insert` and `erase` functions to properly set those parent nodes. Then define a `TreeIterator` class that contains a pointer to a `TreeNode`. The tree's `begin` member function returns an iterator that points to the leftmost leaf. The iterator's `get` member function simply returns the data value of the node to which it points. Its `next` member function needs to find the next element in inorder traversal. If the current node is the left child of the parent, move to the parent. Otherwise, go to the right child if there is one, or to the leftmost descendant of the next unvisited parent otherwise.

P14.13 Implement a tree iterator as described in Exercise P14.2 without modifying the `TreeNode` class. *Hint:* The iterator needs to keep a stack of parent nodes.

P14.14 Write a program that reads a set of floating-point numbers and prints out the ten smallest numbers. As you process the inputs, do not store more than eleven numbers. Insert the numbers into a priority queue. When it holds more than ten values, remove the largest value.

P14.15 This problem illustrates the use of a discrete event simulation, as described in Special Topic 14.3. Imagine you are planning to open a small hot dog stand. You need to determine how many stools your stand should have. Too few stools and you will lose customers; too many and your stand will look empty most of the time.

There are two types of events in this simulation. An arrival event signals the arrival of a customer. If seated, the customer stays a randomly generated amount of time then leaves. A departure event frees the seat the customer was occupying.

Simulate a hotdog stand with three seats. To initialize the simulation a random number of arrival events are scheduled for the period of one hour. The output shows what time each customer arrives and whether they stay or leave. The following is the beginning of a typical run:

```
time 0.13 Customer is seated
time 0.14 Customer is seated
time 0.24 Customer is seated
time 0.29 Customer finishes eating, leaves
time 0.31 Customer is seated
time 0.38 Customer finishes eating, leaves
time 0.41 Customer is seated
time 0.42 Customer is unable to find a seat, leaves
time 0.48 Customer is unable to find a seat, leaves
time 0.63 Customer is unable to find a seat, leaves
time 0.64 Customer is unable to find a seat, leaves
time 0.68 Customer finishes eating, leaves
time 0.71 Customer is seated
```

P14.16 Simulate the processing of customers at a bank with five tellers. Customers arrive on average once per minute, and they need an average of five minutes to complete a transaction. Customers enter a queue to wait for the next available teller.

Use two kinds of events. An arrival event adds the customer to the next free teller or the queue and schedules the next arrival event. When adding a customer to a free teller, also schedule a departure event. The departure event removes the customer from the teller and makes the teller service the next customer in the waiting queue, again scheduling a departure event.

For greater realism, use an exponential distribution for the time between arrivals and the transaction time. If m is the desired mean time and r a uniformly distributed random number between 0 and 1, then $-m \log(r)$ has an exponential distribution.

After each event, your program should print the bank layout, showing empty and occupied tellers and the waiting queue, like this:

```
C.CC.
```

if there is no queue, or

```
CCCCC CCCCCCCCCCCCCCCC
```

if there is one. Simulate customer arrivals for 8 hours. At the end of the simulation, print the total number of customers served and the average time each customer spent in the bank. (Your `Customer` objects will need to track their arrival time.)

P14.17 In most banks, customers enter a single waiting queue, but most supermarkets have a separate queue for each cashier. Modify Exercise P14.16 so that each teller has a separate queue. An arriving customer picks the shortest queue. What is the effect on the average time spent in the bank?

P14.18 Modify the implementation of the `Heap` class so that the parent and child index positions are computed directly, without calling helper functions.

P14.19 Modify the implementation of the `Heap` class so that the `0` element of the array is not wasted.

P14.20 Modify the implementation of the `Heap` class so that it stores strings, not integers. Test your implementation with the tasks from the `pqueue.cpp` program.

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. –4 –3 –2 –1 1 4 9 16
4. 2 4 8 10 14 16
5. It sets `c` to the intersection of `a` and `b`, that is, the set of elements that are present in both `a` and `b`.
6. 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.
7. For example, `Sarah`. Any string between `Romeo` and `Tom` will do.
8. The tree at this point is

```
      D
     /
    B
   /
  A
```

   The call of the `BinarySearchTree::insert` function allocates a node holding the string "`C`", and then calls `TreeNode::insert_node` on the root node (whose data is "`D`"). Since "`C`" < "`D`" and its left node is not `NULL`, the `TreeNode::insert_node` is called on the node whose data is "`B`". Now "`B`" < "`C`", so the second branch of the `if` statement is executed. Since right is `NULL`, right is set to the new node, and the result is

```
      D
     /
    B
   / \
  A   C
```

9. D B A C F E G and A B C D E F G
10.

```
  H
 /
E
```

11.
```
map<string, int> names;
names["one"] = 1; names["two"] = 2; names["three"] = 3; names["four"] = 4;
names["five"] = 5;
```

12.
```
for (map<string, int>::iterator pos = names.begin; pos != names.end(); pos++)
{
```

```
   cout << pos->second << endl;
}
```

13. The iterator visits the key/value pairs in the dictionary order of the keys, that is "five", "four", "one", "three", "two". Therefore the values are printed as 5 4 1 3 2.
14. A `map<string, set<int> >`.
15. If the word has never been seen before, then `word_pages[word]` does not exist. You need to test for that case and insert an empty set:

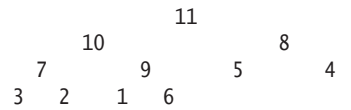
    ```
    if (word_pages.find("word") == word_pages.end()) { word_pages[word] = set<int>(); }
    ```

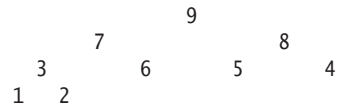
16. A priority queue is appropriate because we want to get the important events first, even if they have been inserted later.
17. Keep a `map<int, string>` that maps priority values to tasks. In each step, get the largest element in the map:

    ```
    map<int, string>::iterator pos = tasks.end();
    pos--;
    ```

    Then display `pos->first` and `pos->second` and remove the entry:

    ```
    tasks.erase(pos);
    ```

18. Priority queues use a heap data structure that is more efficient than the binary search tree used for a map.
19. Yes, but a binary search tree isn't almost filled, so there may be holes in the array.
20. Each insertion or removal is an $O(\log(n))$ operation, and they are repeated n times. Therefore, the efficiency is $O(n \log(n))$
21. It does not require an auxiliary array for storing intermediate values.
22. 

23. 

24. 

    The second one is a max-heap.